ETSI TR 103 528 V1.1.1 (2018-08)

SmartM2M;
Landscape for open source and standards for cloud native software applicable for a Virtualized IoT service layer

Reference
DTR/SmartM2M-103528

Keywords
cloud, IoT, open source, virtualisation

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Contents

List of figures

List of tables

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (https://ipr.etsi.org/).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

Foreword

This Technical Report (TR) has been produced by ETSI Technical Committee Smart Machine-to-Machine communications (SmartM2M).

Modal verbs terminology

In the present document "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

Introduction

In addition to interoperability and security that are two recognized key enablers to the development of large IoT systems, a new one is emerging as another key condition of success: virtualization. The deployment of IoT systems will occur not just within closed and secure administrative domains but also over architectures that support the dynamic usage of resources that are provided by virtualization techniques over cloud back-ends.

This new challenge for IoT requires that the elements of an IoT system can work in a fully interoperable, secure and dynamically configurable manner with other elements (devices, gateways, storage, etc.) that are deployed in different operational and contractual conditions. To this extent, the current architectures of IoT will have to be aligned with those that support the deployment of cloud-based systems (private, public, etc.).

Moreover, these architectures will have to support very diverse and often stringent non-functional requirements such as scalability, reliability, fault tolerance, massive data, security. This will require very flexible architectures for the elements (e.g. the application servers) that will support the virtualized IoT services, as well as very efficient and highly modular implementations that will make a massive usage of Open Source components.

These architectures and these implementations form a new approach to IoT systems and the solutions that this STF will investigate will also have to be validated: to this extent, a Proof-of-Concept implementation involving a massive number of virtualized elements will be made.

The present document is one of three Technical Reports addressing this issue:

- ETSI TR 103 527 [i.1]: "Virtualized IoT Architectures with Cloud Back-ends".
- ETSI TR 103 528 (the present document): "Landscape for open source and standards for cloud native software for a Virtualized IoT service layer".
- ETSI TR 103 529 [i.2]: "Virtualized IoT over Cloud back-ends: A Proof of Concept".

1 Scope

The present document:

- Recalls the main elements of the High-Level Architecture (HLA) in support of IoT Virtualization as it is described in ETSI TR 103 527 [i.1] and how Open Source Software (OSS) and Standards can be used in the implementation of virtualized IoT systems.
- Presents, for each of the layers (and sub-layers) of the HLA, several of the OSS components that have been developed by the open source communities.
- Presents on-going developments in standardization that can be used in support of such implementations.

2 References

2.1 Normative references

Normative references are not applicable in the present document.

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication ETSI cannot guarantee their long-term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] ETSI TR 103 527: "SmartM2M; Virtualized IoT Architectures with Cloud Back-ends".

[i.2] ETSI TR 103 529: "SmartM2M; IoT over Cloud back-ends: a Proof of Concept".

[i.3] ITU-T News, October 2017: "What is 'cloud-native IoT' and why does it matter?".

NOTE: Available at http://news.itu.int/what-is-cloud-native-iot-why-does-it-matter/.

[i.4] "Cloud Native Infrastructure", Justin Garrison, Kris Nova, O'Reilly Media, 2018.

[i.5] ETSI TR 103 375: "SmartM2M; IoT Standards landscape and future evolutions".

[i.6] ETSI TR 103 376: "SmartM2M IoT LSP use cases and standards gaps".

[i.7] ETSI SR 003 392: "Cloud Standards Coordination Phase 2; Cloud Computing Standards Maturity Assessment; A new snapshot of Cloud Computing Standards".

[i.8] ETSI TS 103 264 (V2.1.1) (03-2017): "SmartM2M; Smart Appliances; Reference Ontology and oneM2M Mapping".

[i.9] White Paper: "IoT Platforms Interoperability Approaches", IoT-EPI Platforms Interoperability Task Force, 2017.

[i.10] NIST SP 1500-1: "NIST Big Data Interoperability Framework: Volume 1, Definitions".

[i.11] NIST SP 1500-2: "NIST Big Data Interoperability Framework: Volume 2, Big Data Taxonomies".

[i.12] NIST SP 1500-3: "NIST Big Data Interoperability Framework: Volume 3, Use Cases and General Requirements".

[i.13] NIST SP 1500-4: "NIST Big Data Interoperability Framework: Volume 4, Security and Privacy".

[i.14] NIST SP 1500-5: "NIST Big Data Interoperability Framework: Volume 5, Architectures White Paper Survey".

[i.15] NIST SP 1500-6: "NIST Big Data Interoperability Framework: Volume 6, Reference Architecture".

[i.16] NIST SP 1500-7: "NIST Big Data Interoperability Framework: Volume 7, Standards Roadmap".

[i.17] Recommendation ITU-T Y.4100 (former Y.2066): "Common requirements of the Internet of things".

[i.18] ISO/IEC DIS 20546: "Information technology -- Big data -- Overview and vocabulary".

[i.19] Recommendation ITU-T Y.4114: "Specific requirements and capabilities of the Internet of things for big data".

[i.20] Recommendation ITU-T Y.2068: "Functional framework and capabilities of the Internet of things".

[i.21] ISO/IEC 20547: "Information technology -- Big data reference architecture -- Part 4: Security and privacy".

3 Definitions and abbreviations

3.1 Definitions

For the purposes of the present document, the following terms and definitions apply:

Open Source Software (OSS): computer software that is available in source code form

NOTE: The source code and certain other rights normally reserved for copyright holders are provided under an open-source license that permits users to study, change, improve and at times also to distribute the software.

source code: any collection of computer instructions written using some human-readable computer language, usually as text

standard: output from an SSO

Standards Setting Organization (SSO): any entity whose primary activities are developing, coordinating, promulgating, revising, amending, reissuing, interpreting or otherwise maintaining standards that address the interests of a wide base of users outside the standards development organization

NOTE: In the present document, SSO is used equally for both Standards Setting Organization or Standards Developing Organizations (SDO).

3.2 Abbreviations

For the purposes of the present document, the following abbreviations apply:

AIOTI	Alliance for IoT Innovation
AM	Application Master
API	Application Programming Interface
AWS	Amazon Web Services
BASH	Bourne-again shell (a Unix Shell)
CA	Certificate Authority
CDN	Content Delivery Network
CLI	Command Line Interface
CNCF	Cloud Native Computing Foundation

CPU	Central Processing Unit
CRUD	create, read, update, delete
CSC	Cloud Standards Coordination
CSP	Cloud Service Provider
CSV	Comma-Separated Values
DAG	Directed Acyclic Graph
DC/OS	Datacenter Operating System
DDL	Data Definition Language
DIS	Draft International Standard (in ISO)
DNS	Domain Name Service
DVC	Desktop Cloud Visualization
EBS	Elastic Block Store
EC2	Elastic Compute Cloud
ECR	EC2 Container Registry
ECS	EC2 Container Service
EFS	Elastic File System
ELB	Elastic load Balancing
EMR	Elastic MapReduce
ENA	Elastic Network Adapter
ES	ElasticSearch Service
ETS	Elastic Transcoder Service
EU	European Union
FTP	File Transfer Protocol
GPL	GNU General Public License
GSI	Global Secondary Index
GUI	Graphical User Interface
HDFS	Hadoop Distributed File System
HLA	High-Level Architecture
HPE	Hewlett Packard Enterprise
HSM	Hardware Security Module
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPD	HTTP Daemon
HTTPS	HyperText Transfer Protocol Secure
IaaS	Infrastructure as a Service
IAM	Identity and Access Management
ICMP	Internet Control Message Protocol
IEC	International Electrotechnical Commission
IO	Input/Output
IP	Internet Protocol
ISO	International Organization for Standardization
IT	Information Technology
ITU-T	ITU Telecommunication Standardisation Sector
JAR	Java Archive
JDBC	Java Database Connectivity
JMX	Java Management Extensions
JSON	JavaScript Object Notation
JTC	Joint Technical Committee
JVM	Java Virtual Machine
KMS	Key Management Service
LDAP	Lightweight Directory Access Protocol
LRU	Least Recently Used
MB	Megabyte
NBDIF	NIST Big Data Interoperability Framework
NIST	National Institute of Standards and Technology
OCI	Open Container Image
OLAP	Online Analytical Processing
OSS	Open Source Software
PaaS	Platform as a Service
PHP	Hypertext Preprocessor
RAM	Random Access Memory
RDF	Resource Description Framework

RDS	Relational Database Service
REST	Representational State Transfer
RM	Resource Manager
RPC	Remote Procedure Call
RTT	Round-Trip Time
SaaS	Software as a Service
SAREF	Smart Appliance Reference Ontology
SDO	Standards Development Organization
SES	Simple Email Service
SLA	Service-Level Agreement
SMS	Short Message Service
SNS	Simple Notification Service
SP	Special Publication (of NIST)
SQL	Structured Query Language
SQS	Simple Queue Service
SR	Special Report
SSH	Secure Shell
SSO	Standards Setting Organization
STF	Specialist Task Force
SWF	Simple Workflow
TCP	Transmission Control Protocol
TFS	Team Foundation Server
TLS	Transport Layer Security
TPM	Trusted Platform Module
TRL	Technology Readiness Level
TSV	Tab Separated Values
UDP	User Datagram Protocol
UI	User Interface
URL	Uniform Resource Locator
VM	Virtual Machine
VPC	Virtual Private Cloud
VPN	Virtual Private Network
W3C	World-Wide Web Consortium
WG	Work Group
XML	Extensible Markup Language

4 A Landscape for Open Source and Standards

4.1 Introduction

The IoT industry starts to understand the potential benefits of Cloud-Native Computing for the fast, effective and future-safe development of IoT systems combining the strengths of both IoT and Cloud industries in a new value proposition (see [i.3] for example).

The notion of Cloud-Native Computing is now widely supported by a large set of technologies embedded in Cloud-Native Infrastructures in support of Cloud-Native Applications. A possible definition is: "Cloud native infrastructure is infrastructure that is hidden behind useful abstractions, controlled by APIs, managed by software, and has the purpose of running applications. Running infrastructure with these traits gives rise to a new pattern for managing that infrastructure in a scalable, efficient way" (see [i.4]).

The expectation of Cloud-Native applications is to benefit from offerings from Cloud Service Providers (CSP) that may cover parts or all of the layers of Virtualized application, via Infrastructure as a Service (IaaS), Platform as a Service (PaaS) or Software as a Service (SaaS). Figure 1 presents the possible usages of such offerings in delegating more and more important parts of the underlying layers to a third-party in charge of hiding complexity, resource usage, etc.

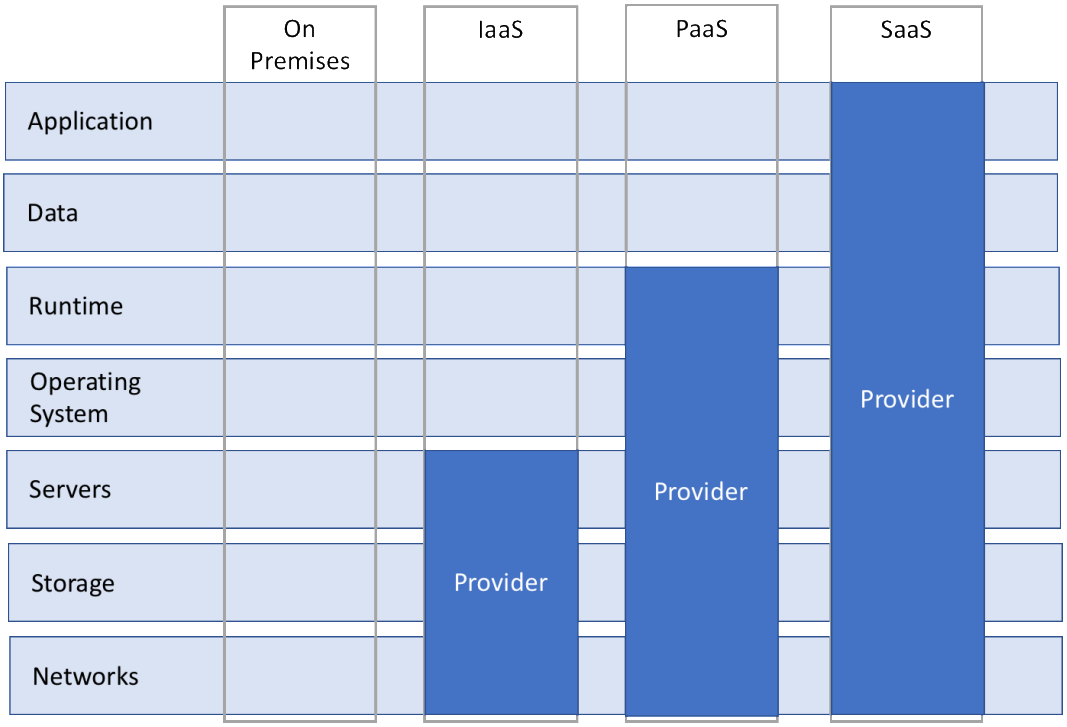

Figure 1: The potential of Cloud-Native Infrastructures

4.2 Open Source Software, Cloud-Native Computing, IoT

In the case of IoT applications, the trade-off between what is delegated to the Cloud Service Provider and what is kept in the hands of the application developers may vary depending a large number of potential factors. It is, for example, not always clear which support of various IoT devices is available in a IaaS offer or how data at the edge can be efficiently handled in a PaaS offer. This is why it is important to investigate the degree of flexibility that is offered to the application developer by complementary solutions such as those provided by the Open Source Software.

The technologies assembled by the Open Source communities around Cloud-Native Computing are now more and more diverse (involving all layers of a Micro-Service Architecture such as the one described in ETSI TR 103 527 [i.1]). In the same time, these technologies have been made much easier to apprehend, to master and to package into more and more complex systems.

It is important, in the context of IoT, where a number of systems are being developed with an important part of "greenfield" development with new features and new devices deployed, to assess how these technologies are available to the application developers. An inventory of major available OSS components is an essential first step.

4.3 Content of the present document

Clause 5 recalls the main architectural elements developed in ETSI TR 103 527 [i.1] and serves as a structured basis for an introduction of OSS components which is done in subsequent Clause 6. Is also presents how these OSS components are supported by OSS communities and eco-systems in order to ensure their perennial use by virtualized IoT systems developers.

Clause 6 presents a detailed description of a number of selected OSS components that are currently used for Cloud-Native implementations and that can be used in IoT Virtualization implementation projects. Many of these OSS components are used in the proof-of-concept implementation developed in ETSI TR 103 529 [i.2].

Clause 7 addresses the support that standardization can bring, in addition to the OSS components, in support of IoT Virtualization implementations. It assesses the current status of standardization in Cloud Computing and IoT and presents potentially useful on-going developments in standardization or pre-standardization.

Clause 8 summarizes the main findings of the present document and provides a set of recommendations for architects and developers in charge of potential IoT virtualization projects.

5 Open Source support to IoT Virtualization

5.1 An Architecture for OSS component classification

A Micro-Service Architecture is presented in ETSI TR 103 527 [i.1] with the purpose of identifying layers in the development of IoT systems that can be mapped with on-going developments in the OSS communities. This Micro-Service Architecture is recalled in Figure 2.

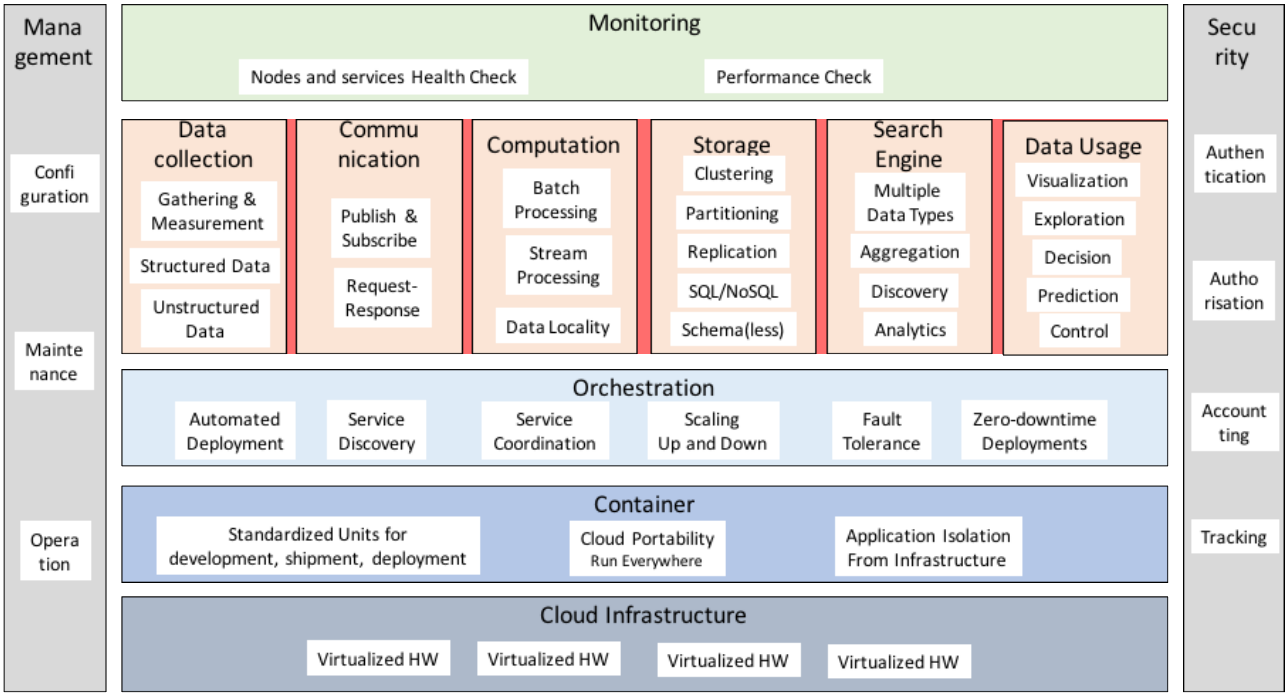

Figure 2: An HLA for IoT Virtualization

An important layer is the one of "Common Services" (i.e. Data Collection, Communication, etc.) where a very large number of (sometimes competing) initiatives are taking place and a wealth of solutions are provided to the applications developers. It is also one of those which will benefit from a detailed and precise map of existing solutions.

5.2 A map of Cloud-Native Software

The OSS communities are well aware of the very large number of initiatives related to Virtualization and the need to provide some form of guidance to those who want to get involved.

An initiative like the Cloud Native Computing Foundation (CNCF, https://www.cncf.io) "builds sustainable ecosystems and fosters a community around a constellation of high-quality projects that orchestrate containers as part of a microservices architecture". More importantly, the CNCF is supporting a Landscape project intended as "a map through the previously uncharted terrain of cloud native technologies". The current landscape of CNCF is shown in Figure 3.

This landscape displays only a few components for each of the software component category presented and is obviously showing a very complex map.

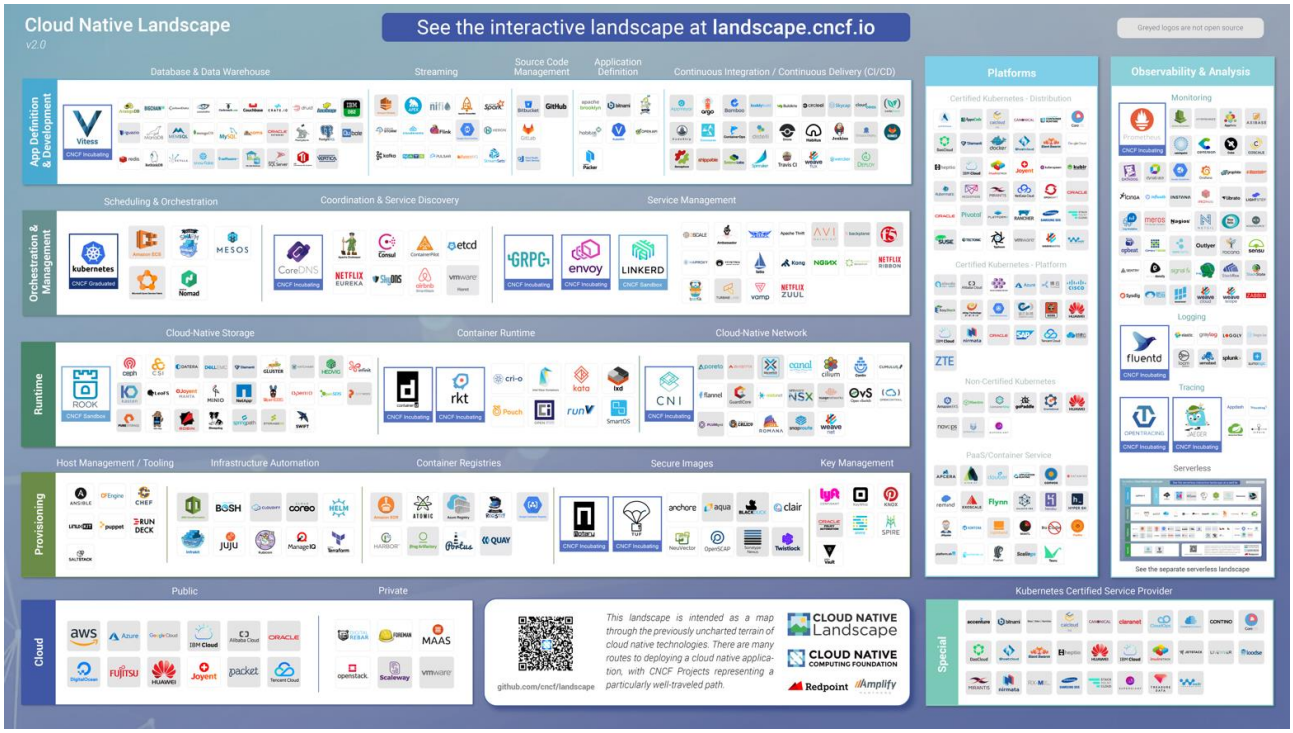

Figure 3: The CNCF landscape of Cloud-Native Software Components

5.3 Open Source Software in support of IoT Virtualization

5.3.1 The approach taken

The present document is offering a more focused approach to the identification of OSS Components in support of IoT Virtualization, essentially for the following reasons:

- The components analysed are not covering the whole of Cloud-Native components and, in particular, are more focused on the "general purpose" cross-domain components, rather than on application-specific ones.
- The selection takes into account a form of "recognition" or of "notoriety" of the selected components and limits the number of choices for each class of component.

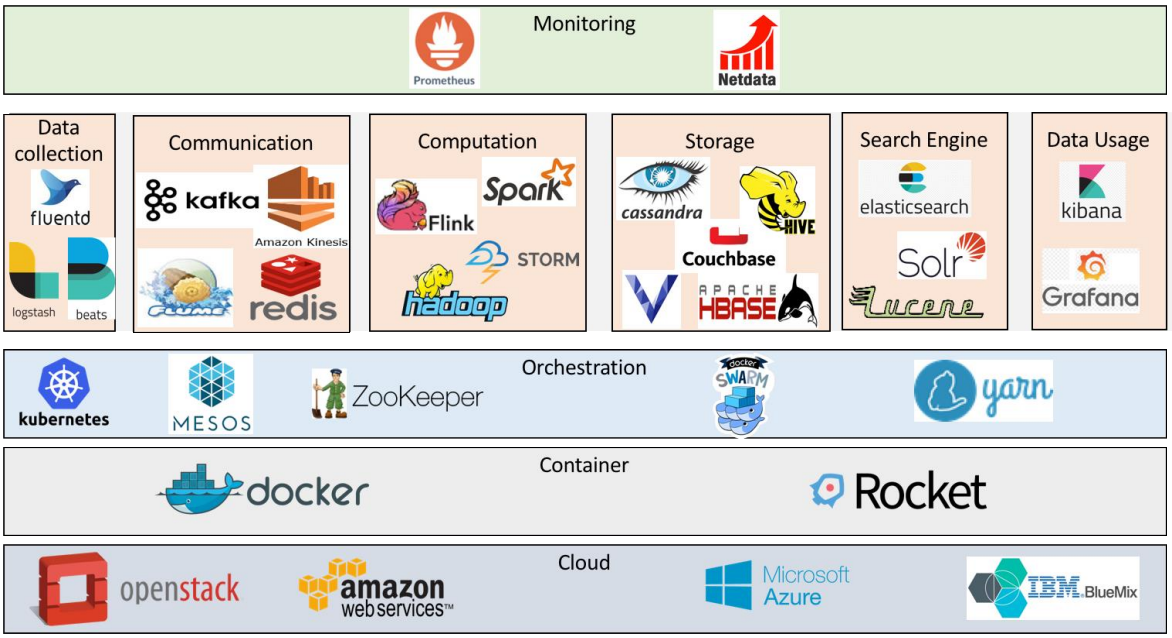

Figure 4: A global map of OSS Components for IoT Virtualization

Altogether, the resulting global map of components in Figure 4 is simpler than the one of Figure 3.

It should be noted that some of the Components presented in the present document are actually used in the implementation of the Proof-of-Concept described in ETSI TR 103 529 [i.2].

5.3.2 The role of Open Source Software eco-systems

In a large number of cases, the OSS components are not developed in isolation of the rest of what the OSS communities are developing. Even if there are thousands of on-going OSS developments concurrently, some become more successful because they are done within the context of large OSS communities supporting the coordinated development of related components. Such "ecosystems" are providing a proper ground to the maturation of components that benefit from the (technical) proximity with a (potentially large) number of other components.

The example of the maturation over time of the "Apache Hadoop" ecosystem in Figure 5 is typical.

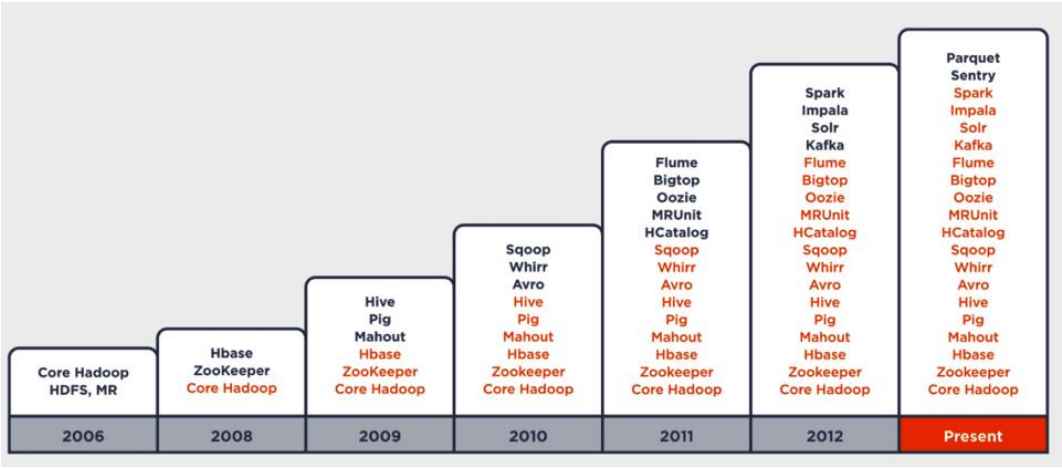

Figure 5: The example of the Apache Hadoop ecosystem

In principle, components belonging to established and recognized eco-systems are more likely to evolve "properly", with more transparent and controlled evolution of features, large support of skilled developers, etc. The potential drawback may be the lack of adaptability of a component to other ecosystems that its ecosystem of origin.

5.3.3 How to read the map

Clause 6 presents, layer by layer, the components shown in Figure 4.For each of the component, the following elements are presented:

- Presentation
- Main concept and features
- Status in the Open Source Community

Readiness: This refers to an evaluation of how the component can be safely used based on how it is developed, evolved, distributed, etc.	**Ecosystem:** Identification of the ecosystem to which the component can be associated	**License:** Which license is used for source code distribution

NOTE: In clause 6, the "Cloud Infrastructure" part (clause 6.1) is slightly different regarding the nature of the components presented. Whereas the whole clause 6 is about Open Source Components, the components presented in clause 6.1 are commercial products. The reason why they are nevertheless presented is that they represent the most widely used components in that space.

6 Open Source Components for IoT Virtualization

6.1 Cloud Infrastructure

6.1.1 OpenStack

Presentation

OpenStack (https://www.openstack.org) is a free and open-source software platform for cloud computing, mostly deployed as Infrastructure-as-a-Service (IaaS). The software platform consists of interrelated components that control diverse, multi-vendor hardware pools of processing, storage, and networking resources throughout a data centre. Users either manage it through a web-based dashboard, through command-line tools, or through a RESTful API.

Main concepts and features

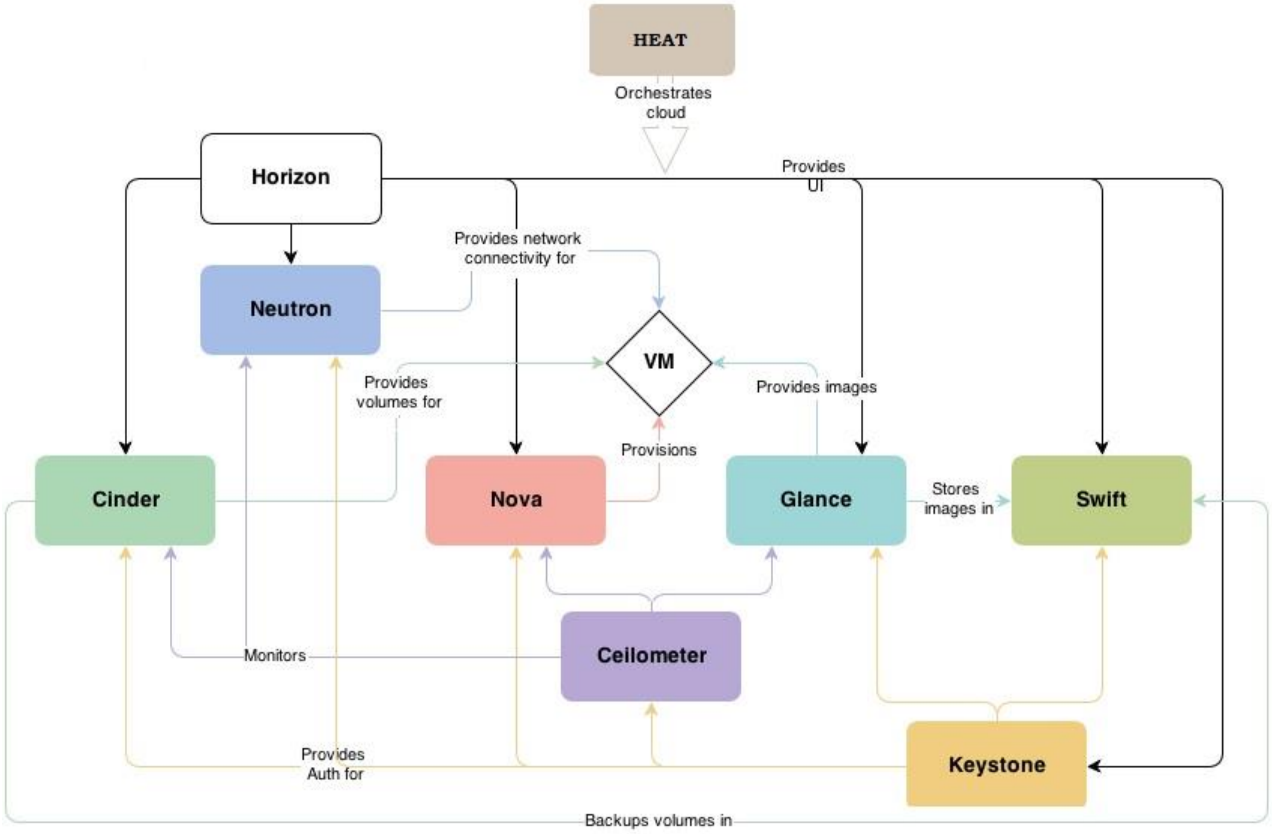

Figure 6: OpenStack architecture

OpenStack offers a modular architecture. The following list describes its Core components:

- **Nova** (https://docs.openstack.org/nova/latest/) provides a way to provision compute instances (aka virtual servers). It supports creating virtual machines, Bare-Metal servers (i.e. single-tenant physical servers) through the use of Ironic (see below), and has limited support for system containers;
- **Neutron** (https://docs.openstack.org/keystone/latest/) provides "network connectivity as a service" between interface devices managed by other OpenStack services. The service works by allowing users to create their own networks and then attach interfaces to them;
- **Glance** (https://docs.openstack.org/glance/latest/) is an image service that provides virtual machine (VM) images discovery, registration, and retrieval. It has a RESTful API that allows querying of VM image metadata as well as retrieval of the actual image;
- **Keystone** (https://docs.openstack.org/keystone/latest/) is an identity service that provides API client authentication, service discovery, and distributed multi-tenant authorization;
- **Swift** (https://docs.openstack.org/swift/latest/) is an object store service that allow efficient, safe, and cheap storage or retrieval of data and files but not mount directories like a fileserver;
- **Horizon** (https://docs.openstack.org/horizon/latest/) provides a modular web-based user interface for all the OpenStack services. This dashboard enables performing most operations on the cloud like launching an instance, assigning IP addresses and setting access controls;
- **Cinder** is a Block Storage service that virtualizes the management of block storage devices and provides end users with a self-service API to request and consume those resources without requiring any knowledge of where their storage is actually deployed or on what type of device;
- **Heat** is an orchestration engine that provides the ability to launch multiple composite cloud applications based on templates in the form of text files that can be treated like code;

- **Ceilometer** is a data collection service that provides the ability to normalize and transform data across all current OpenStack core components. Its data can be used to provide customer billing, resource tracking, and alarming capabilities across all OpenStack core components.

In addition to these core components, there are also several non-core components such as **Trove** (database), **Zaqar** (Messaging), **Barbican** (Key Manager), **Congress** (governance), **Sahara** (Elastic Map reduce), **Manila** (shared file system), **Magnum** (containers), **Ironic** (Bare-Metal provisioning), **Designate** (DNS), **Murano** (application catalog), etc.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** OpenStack	**License:** Apache 2.0

6.1.2 Amazon™ Web Services (AWS)

Presentation

Amazon™ Web Services (AWS) (https://aws.amazon.com) is a secure cloud services platform, offering computing power, database storage, content delivery and other functionality, delivered as a utility: on-demand, available in seconds, with pay-as-you-go pricing. The technology allows subscribers to have at their disposal a full-fledged virtual cluster of computers, available all the time, through the Internet. Based on what the subscribers need and pay for, they can reserve a single virtual AWS computer, a cluster of virtual computers, a physical computer dedicated for their exclusive use, or even a cluster of dedicated physical computers.

Main concepts and features

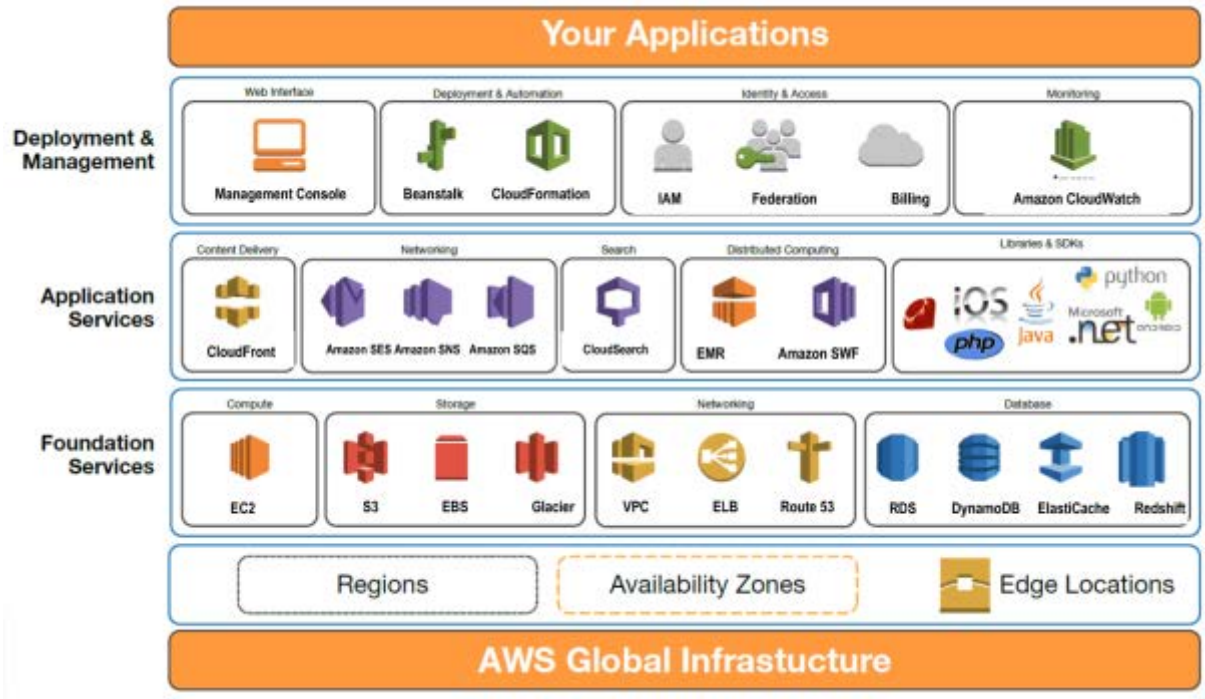

Figure 7: Amazon™ Web Services Architecture

A list of Amazon™ Web services products organized by domains is provided below.

Compute

- Amazon™ **Elastic Compute Cloud** (EC2) is a IaaS service providing virtual servers controllable by an API, based on the Xen hypervisor;

- Amazon™ **Elastic Beanstalk** provides a PaaS service for hosting applications. Equivalent services include Google™ App Engine or Heroku or OpenShift for on-premises use;
- Amazon™ **Lambda** server-less computing platform runs code in response to AWS internal or external events, such as http requests, and transparently provides the resource required.

Networking

- Amazon™ **Route 53** provides a scalable Managed DNS service providing Domain Name Services;
- Amazon™ **Virtual Private Cloud** (VPC) creates a logically isolated set of AWS resources which can be connected using a VPN connection. This competes against on-premises solutions such as OpenStack or HPE Helion Eucalyptus used in conjunction with PaaS software;
- AWS **Direct Connect** provides dedicated network connections into AWS data centres;
- Amazon™ **Elastic Load Balancing** (ELB) automatically distributes incoming traffic across multiple Amazon EC2 instances;
- AWS **Elastic Network Adapter** (ENA) provides up to 25 Gbit/s of network bandwidth to an Amazon™ EC2 instance.

Content Delivery

- Amazon™ **CloudFront** is a content delivery network (CDN) for distributing objects to so-called "edge locations" near the request.
- Amazon™ **Connect** is a self-service, cloud-based contact centre service available to business.
- Amazon™ **Simple Storage Service** (S3) provides scalable object storage accessible from a Web Service interface;
- Amazon™ **Glacier** provides longer-term storage options (compared to S3) with high redundancy and availability, but for low-frequency access times. It is intended for archiving data;
- AWS **Storage Gateway** is an iSCSI block storage virtual appliance with cloud-based backup;
- Amazon™ **Elastic Block Store** (EBS) provides persistent block-level storage volumes for EC2;
- AWS **Import/Export** accelerates moving large amounts of data into and out of AWS using portable storage devices for transport;
- Amazon™ **Elastic File System** (EFS) is a file storage service for Amazon Elastic Compute Cloud (Amazon™ EC2) instances.
- Amazon™ **DynamoDB** provides a scalable, low-latency NoSQL online Database Service backed by SSDs;
- Amazon™ **ElastiCache** provides in-memory caching for web applications. This is Amazon's implementation of Memcached and Redis;
- Amazon™ **Neptune** provides a full-managed graph database service. It supports open graph APIs for both Gremlin and SPARQL;
- Amazon™ **Relational Database Service** (RDS) provides scalable database servers with MySQL, Oracle, SQL Server, and PostgreSQL support;
- Amazon™ **Redshift** provides petabyte-scale data warehousing with column-based storage and multi-node compute;
- Amazon™ **SimpleDB** allows developers to run queries on structured data. It operates in concert with EC2 and S3;

- AWS **Data Pipeline** provides reliable service for data transfer between different AWS compute and storage services (e.g. Amazon™ S3, Amazon™ RDS, Amazon™ DynamoDB, Amazon™ EMR). In other words, this service is simply a data-driven workload management system, which provides a management API for managing and monitoring of data-driven workloads in cloud applications;
- Amazon™ **Aurora** provides a MySQL-compatible relational database engine that has been created specifically for the AWS infrastructure, and that claims faster speeds and lower costs that are realized in larger databases.
- AWS **Mobile Hub** is used for adding and configuring mobile app features, including authentication, data storage, backend logic, push notifications, content delivery, and analytics;
- Amazon™ **Cognito** for adding user sign-up and sign-in to mobile and web apps;
- AWS **Device Farm** is an app testing service for testing and interacting with Android, iOS, and web apps on many devices at once, or reproduce issues on a device in real time.
- Amazon™ **Pinpoint** helps to engage customers via email, SMS and Mobile Push messages, tracking overall customer and engagement activity.
- AWS **CloudFormation** provides a declarative template-based Infrastructure as Code model for configuring AWS;
- AWS **Elastic Beanstalk** provides deployment and management of applications in the cloud;
- AWS **OpsWorks** provides configuration of EC2 services using Chef;
- AWS **CodeDeploy** provides automated code deployment to EC2 instances.
- AWS **Systems Manager** provides visibility and control of infrastructure on AWS and on-premises through a unified user interface to view operational data from multiple AWS services and automate operational tasks across AWS resources. Common operational tasks include remote administration without SSH, secrets management, collecting software inventory, automated patching, and configuration management;
- Amazon™ **Identity and Access Management** (IAM) is an implicit service, providing the authentication infrastructure used to authenticate access to the various services;
- AWS **Directory Service** is a managed service that allows connection to AWS resources with an existing on-premises Microsoft Active Directory or to set up a new, stand-alone directory in the AWS Cloud;
- Amazon™ **CloudWatch**, provides monitoring for AWS cloud resources and applications, starting with EC2;
- AWS **Management Console** (AWS Console) is a web-based point and click interface to manage and monitor the Amazon™ infrastructure suite including (but not limited to) EC2, EBS, S3, SQS, etc.;
- Amazon™ **CloudHSM** is a service that helps to meet corporate, contractual and regulatory compliance requirements for data security by using dedicated Hardware Security Module (HSM) appliances within the AWS cloud.
- AWS **Key Management Service** (KMS) is a managed service to create and control encryption keys;
- Amazon™ **EC2 Container Service** (ECS) is a highly scalable and fast container management service using Docker containers;
- Amazon™ **EC2 Container Registry** (ECR) stores, manages, and deploys Docker container images;
- Amazon™ **Fargate** allows users to run containers without having to manage servers or clusters.

Application services

- Amazon™ **API Gateway** is a service for publishing, maintaining and securing web service APIs;
- Amazon™ **CloudSearch** provides basic full-text search and indexing of textual content;
- Amazon™ **DevPay** (currently in limited beta version) is a billing and account management system for applications that developers have built atop Amazon Web Services;

- Amazon™ **Elastic Transcoder Service** (ETS) provides video transcoding of S3 hosted videos, marketed primarily as a way to convert source files into mobile-ready versions;
- Amazon™ **Simple Email Service** (SES) provides bulk and transactional email sending;
- Amazon™ **Simple Queue Service** (SQS) provides a hosted message queue for web applications;
- Amazon™ **Simple Notification Service** (SNS) provides a hosted multi-protocol "push" messaging for applications;
- Amazon™ **Simple Workflow** (SWF) is a workflow service for building scalable, resilient applications;
- Amazon™ **Cognito** is a user identity and data synchronization service that securely manages and synchronizes app data for users across their mobile devices;
- Amazon™ **AppStream** 2.0 is a low-latency service that streams and resources intensive applications and games from the cloud using NICE's DVC technology.
- Amazon™ **Athena** is an ETL-like service that allows server-less querying of S3 content using standard SQL;
- Amazon™ **Elastic MapReduce** (EMR) Provides a PaaS service delivering Hadoop for running MapReduce queries framework running on the web-scale infrastructure of EC2 and Amazon S3;
- Amazon™ **Machine Learning** is a service that assists developers of all skill levels to use machine learning technology;
- Amazon™ **Kinesis** is a cloud-based service for real-time data processing over large, distributed data streams. It streams data in real time with the ability to process thousands of data streams on a per-second basis;
- Amazon™ **Elasticsearch Service** provides fully managed Elasticsearch and Kibana services;
- Amazon™ **QuickSight** is a business intelligence, analytics, and visualization tool that provides ad-hoc services by connecting to AWS or non-AWS data sources.
- Amazon™ **Sagemaker** is an integrated deep learning development and deployment platform.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** AWS	**License:** Commercial (with some open source projects)

6.1.3 Microsoft™ Azure

Presentation

Microsoft™ Azure (https://azure.microsoft.com) is a cloud computing service created by Microsoft™ for building, testing, deploying, and managing applications and services through a global network of Microsoft-managed datacentres. It provides software as a service (SaaS), platform as a service (PaaS) and infrastructure as a service (IaaS) and supports many different programming languages, tools and frameworks, including both Microsoft-specific and third-party software and systems.

Main concepts and features

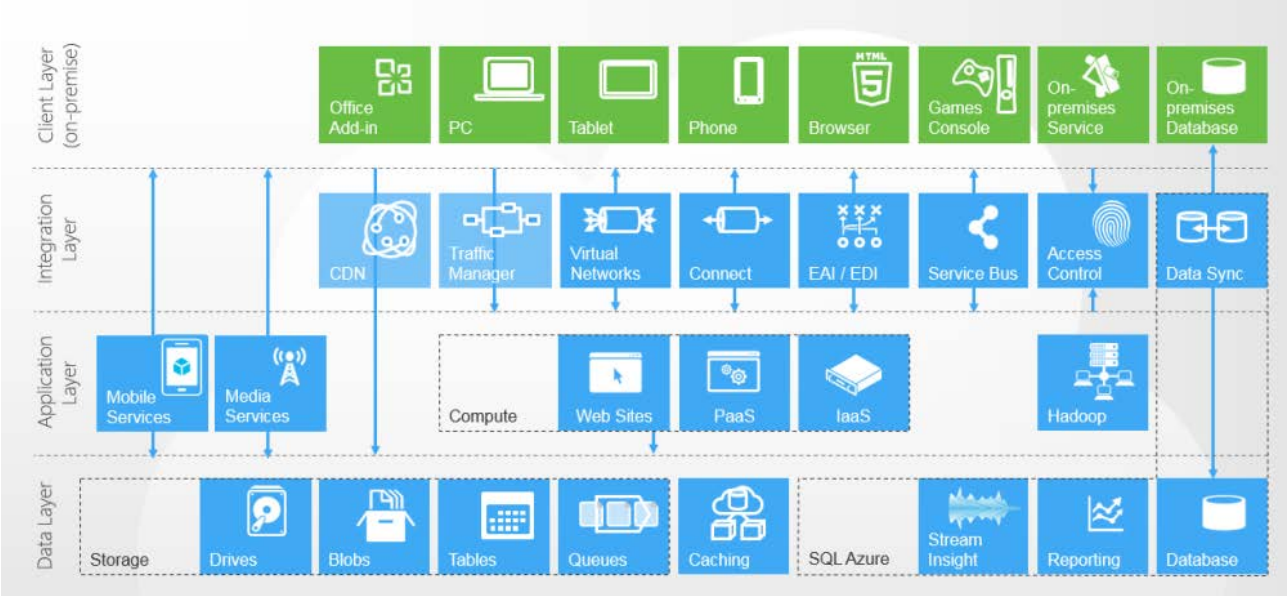

Figure 8: Microsoft™ Azure Architecture

Microsoft™ Azure lets developers and IT workers easily create and deploy modern, cross-platform web and mobile applications. The system also allows to store information in the cloud and back it up for quick and easy recovery.

Azure runs enterprise applications as well as large scale computing processes and implements powerful predictive analytics. It also enables users to create intuitive products and services by leveraging Internet of Things services. Azure operates on a global network of Microsoft-managed datacentres spanning 22 regions.

The main advantages of using Microsoft™ Azure are:

- Users can start for and scale up as traffic improves. Build with Node.js, PHP, or ASP.NET and deploy in seconds with TFS, Git, or FTP;
- Users can easily deploy and run Linux and Windows Server virtual machines. In addition, they can migrate apps and infrastructure without changing existing code;
- Push notifications, user authentication, and structured storage can be incorporated in minutes;
- Azure is an Enterprise-grade cloud platform using a rich PaaS environment on top of which automated deployments and multi-tier scenarios are supported;
- Azure PaaS solution makes use of a fully compatible enterprise-ready Hadoop service, offers easy management and integrates with System Centre and Active Directory;
- Users can create, manage and distribute media in the cloud with a solution that offers everything from content protection to encoding to analytics support to streaming.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Microsoft	**License:** Commercial

6.1.4 IBM™ BlueMix

Presentation

IBM™ Bluemix (https://www.ibm.com/cloud) is a cloud platform as a service (PaaS) developed by IBM™. It supports several programming languages and services as well as integrated DevOps to build, run, deploy and manage applications on the cloud. Bluemix supports several programming languages including Java, Node.js, Go, PHP, Swift, Python, Ruby Sinatra, Ruby on Rails and can be extended to support other languages such as Scala through the use of buildpacks.

Main concepts and features

Bluemix provides three compute technologies: **Cloud Foundry**, **Docker** and **OpenStack**. So, the applications can run respectively using instant runtimes, containers or virtual machines when users can pick the level of infrastructure for architectural applications needs.

IBM™ Bluemix exposes three ways to deploy applications: Bluemix Public, Bluemix Dedicated (which are both powered by IBM™ SoftLayer) or Bluemix Local running within the user datacentre. **SoftLayer** is an IBM's Bluemix IaaS, providing a seamlessly unified global cloud computing infrastructure. It combines virtual public cloud instances, powerful bare-metal servers, turnkey private clouds, and a broad range of storage, network and security devices and services. The IBM™ SoftLayer datacentres are intended to meet growing needs in terms of choice, compliance, and data residency.

IBM™ Bluemix combines the application's development with its lifecycle management. It exposes operations such as management, testing, configuration and software maintenance by leveraging some of the capabilities of DevOps Services. It offers ready-to-use services that are created whether by IBM™ or business partners to enhance the functionality of an application. It is capable to scale the performance as workload increases. Scalability is ensured through the quick provision of SoftLayer infrastructure. It is Based on an open-source cloud computing technologies (Cloud Foundry, Docker and OpenStack).

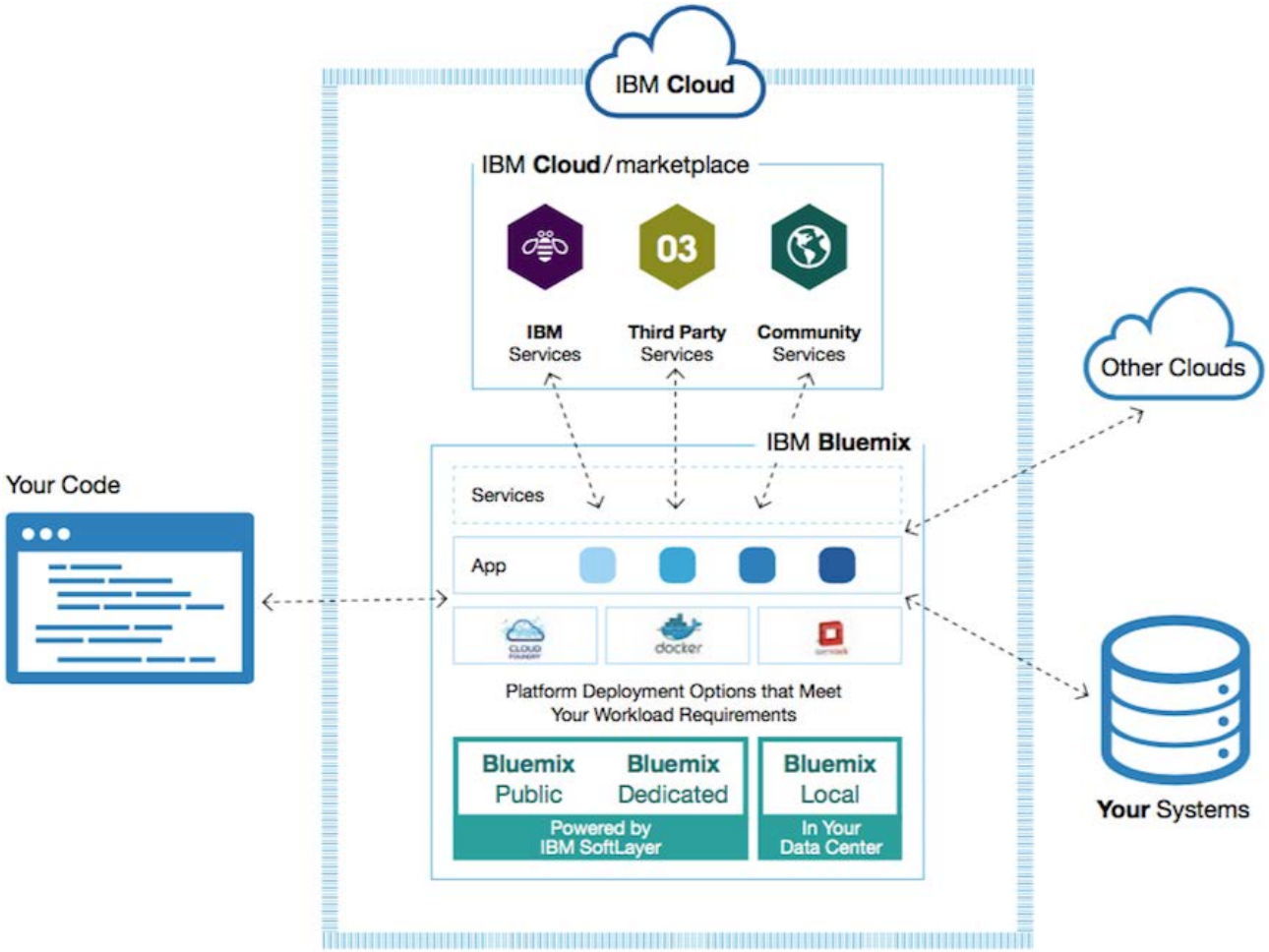

Figure 9: IBM Bluemix Architecture

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** IBM	**License:** Commercial

6.2 Container

6.2.1 Docker

Presentation

Docker (https://www.docker.com/) is an open source project that automates the deployment of applications inside software containers. Docker implements a high-level API to provide lightweight containers that run processes in isolation. Built on top of facilities provided by the Linux kernel, a Docker container, unlike a virtual machine, does not require or include a separate operating system. Instead, it relies on the kernel's functionality, uses resource isolation (CPU, memory, block I/O, network, etc.) and separates namespaces to isolate the application's view of the operating system. It also allows to deploy and scale more easily since Docker containers can run almost everywhere: on desktops, physical servers, virtual machines, into datacentres, in public or private clouds.

Main concepts and features

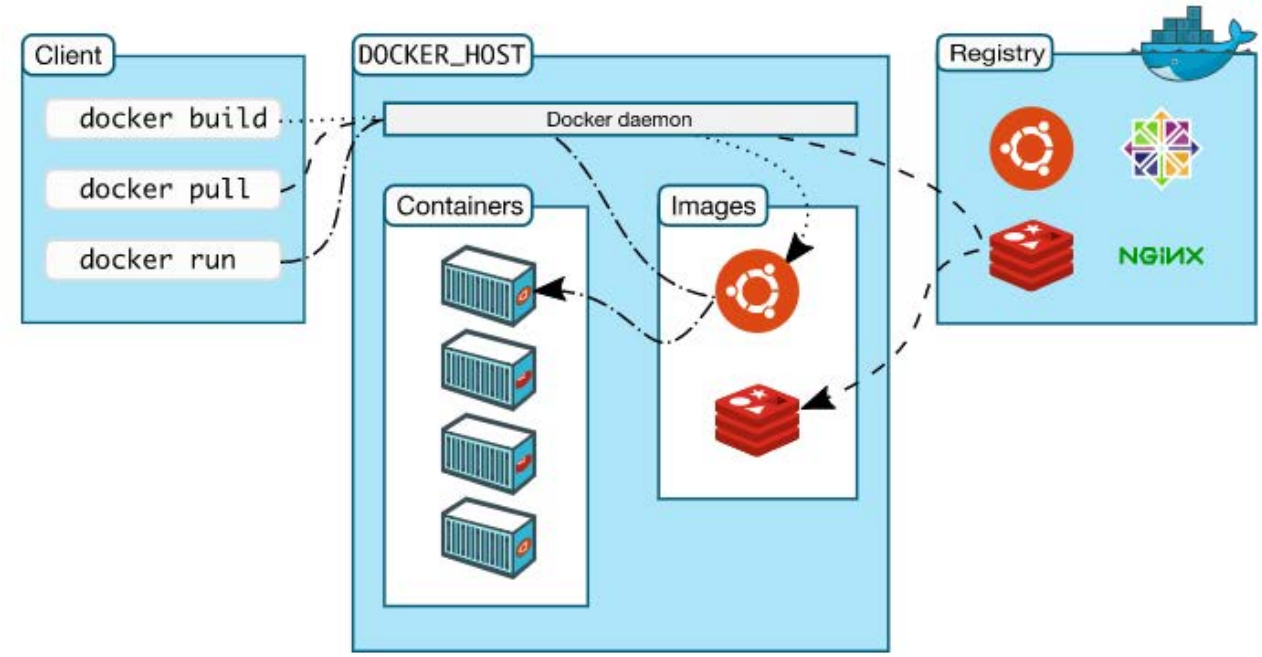

Figure 10: Docker Architecture

Docker uses a client-server architecture where Docker client and Docker daemon communicate through a RESTful API. The daemon operates and distributes the Docker containers making most of the work of building. The Docker client is the primary user interface to Docker, it accepts commands from the user and communicates with the Docker daemon.

The top-level components of Docker are:

- **Docker Images**: an image is a read-only template with instructions for creating a Docker container.
- **Docker Container**: a container is a runnable instance of an image. One can create, run, stop, move, or delete a container using the Docker API.
- **Docker Registries** are public or private stores from which one can upload or download images. **Docker Hub** is the public Docker registry, and it provides a huge collection of existing images. These images can be created by the user or can be images that others have previously created. Docker registries are the distribution component of Docker.
- **Docker Engine** is a lightweight and powerful open source containerization technology combined with a work flow for building and containerizing one's applications.
- **Dockerfiles** are scripts containing a successive series of instructions, directions, and commands which will be executed to form a new docker image. The Dockerfiles replace the process of doing everything manually and repeatedly.

Some of Docker features are:

- Faster delivery of applications: containers have sub-second launch times, reducing the cycle time of development, testing, and deployment.
- Easy deployment and scale: Docker containers run everywhere on many platforms.
- Docker containers do not need a hypervisor, which allows to pack more of them onto hosts, and get higher density.
- As Docker speeds up the workflow, it gets easier to make lots of minor changes instead of huge updates. Smaller changes mean reduced risk and more uptime. Faster deployment makes for easier management.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Docker	**License:** Apache 2.0

6.2.2 Rocket

Presentation

Rocket (https://coreos.com/rkt/) (**rkt**) is an open source software for container management for Linux clusters. Designed for security, simplicity, and composability within modern cluster architectures, rkt discovers, verifies, fetches, and executes application containers with pluggable isolation. Rkt's primary interface comprises a single executable, rather than a background daemon, and rkt leverages this design to easily integrate with existing init systems, like systemd, and with advanced cluster orchestration environments, like Kubernetes. rkt implements a modern, open, standard container format, the App Container (appc), but can also execute other container images, like those created with Docker.

Main concepts and features

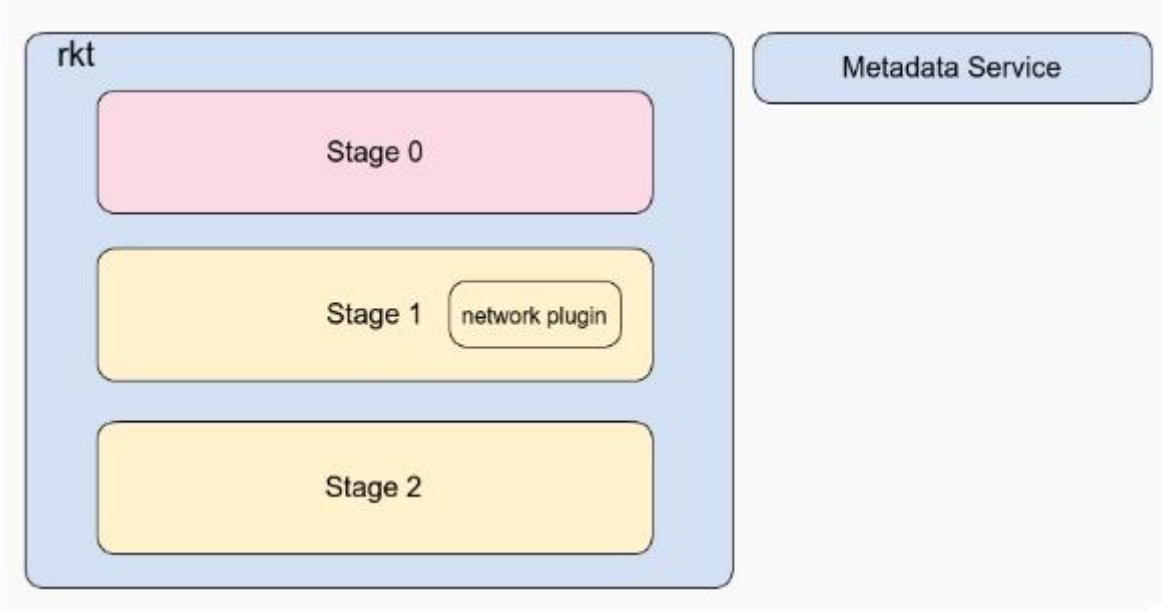

Figure 11: Rocket Architecture

Some of rkt's key features and goals are:

- The basic unit of execution of rkt is a pod, linking together resources and user applications in a self-contained environment.
- Security: rkt is developed with a principle of "secure-by-default". It includes a number of important security features, TPM measurement, and running app containers in hardware-isolated VMs.
- Composability: rkt is designed for first-class integration with init systems (like systemd, upstart) and cluster orchestration tools and supports swappable execution engines.
- Open standards and compatibility: rkt implements the appc (App Container) specification, supports the Container Networking Interface specification, and can run Docker images and OCI images. A broader native support for OCI images and runtimes is under development.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** RedHat (CoreOS) and Cloud Native Computing Foundation	**License:** Apache 2.0

6.2.3 Comparison of Container Software

Both Docker and rkt are used for creating containerized applications. The comparison of two technologies is discussed in table 1.

Table 1: Comparison of Container Software

Criteria	Docker	Rkt
Smaller Unit	Node	Pod
Network model	Container Network model	Container Network Interface.
Network	• Runs with super-user privileges and spins off new containers as its sub-process: Vulnerability.	• New containers are never created from a root privileged process: no root escalation attacks.
Composability	Requires custom in-container init systems to manage child processes.	Proper unix process model, manage processes with system, standard sysv init, runit, etc.
Flexibility in publishing and sharing images	Need to setup a special private registry in user' servers or host it in a Docker paid account.	Need a web server and uses HTTPS protocol to download images and uses a meta description on the web server to point to the location.
Security	SELinux, Seccomp, Apparmor.	SELinux, Seccomp, TPM.
Cluster	Swarm, Kubernetes, Nomad, Fleet, Cloud Foundry.	Fleet, Kubernetes, Nomad.
Filesystem	Overlayfs, Aufs, Brtfs, ZFS plugins.	Oberlayfs, Aufs.
Community support	Large and active community	Negligible community
Notoriety	Used by many leading companies such as: Google, Amazon, Paypal, Ebay, Spotify and Uber.	Used by smaller companies: Verizon, Salesforce.com, DigitalOcean.

6.3 Orchestration

6.3.1 Kubernetes

Presentation

Kubernetes (https://kubernetes.io/) is an open source system for automating the deployment, scaling, and management of containerized applications. It groups the containers that make up an application into logical units for easy management and discovery. It orchestrates computing, networking, and storage infrastructure on behalf of user workloads. This provides much of the simplicity of Platform as a Service (PaaS) with the flexibility of Infrastructure as a Service (IaaS), and enables portability across infrastructure providers.

Main concepts and features

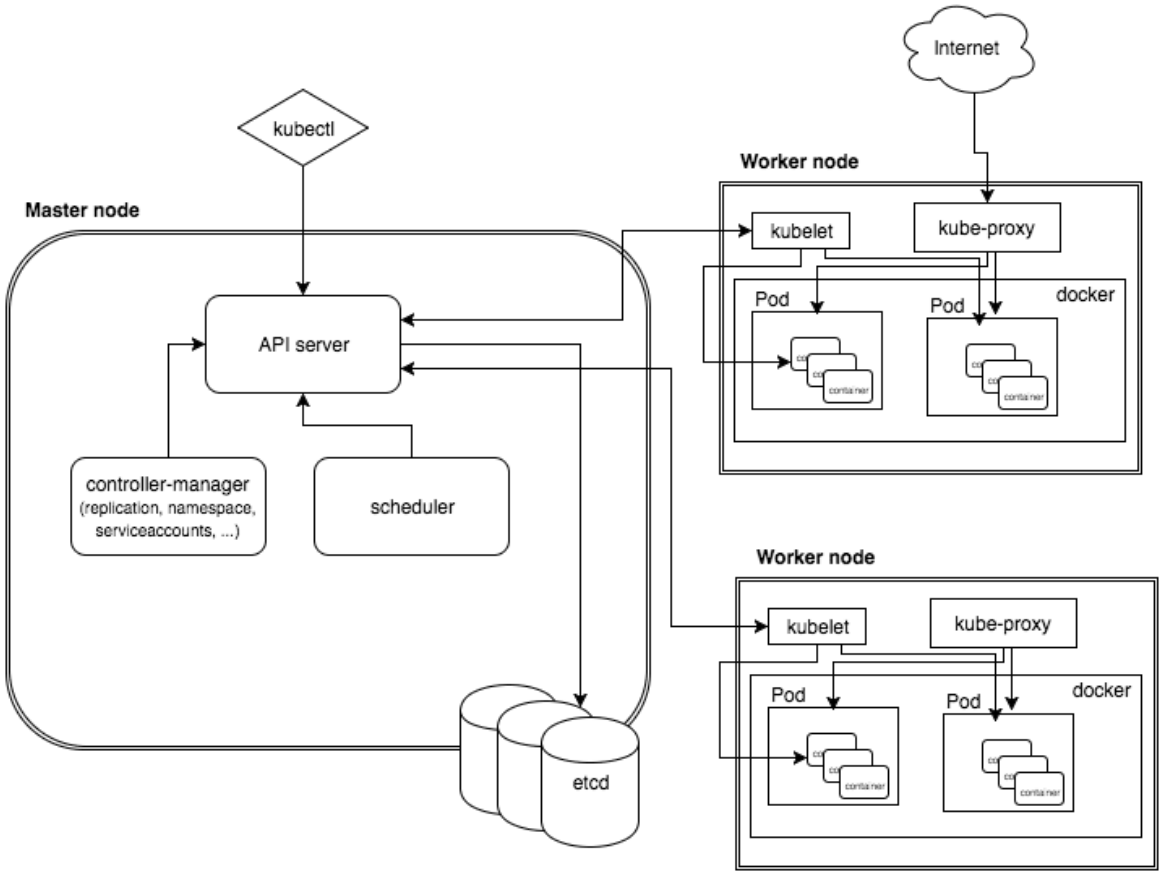

Figure 12: Kubernetes architecture

Kubernetes architecture provides a flexible, loosely-coupled mechanism for service discovery. Like most distributed computing platforms, a Kubernetes cluster consists of at least one master and multiple compute nodes.

- **Master Node:** The master node is responsible for the management of Kubernetes cluster. This is the entry point of all administrative tasks. The master node is the one taking care of orchestrating the worker nodes, where the actual services are running:
  - **API server:** is the entry point for all the REST commands used to control the cluster. It processes the REST requests, validates them, and executes the attached business logic. The result state will be persisted somewhere, which is the role of the following component of the master node.
  - **etcd storage:** is a simple, distributed, consistent key-value store. It is mainly used for shared configuration and service discovery. It provides a REST API for CRUD operations as well as an interface to register watchers on specific nodes, which enables a reliable way to notify the rest of the cluster about configuration changes.
  - **Scheduler:** is responsible of the deployment of configured pods and services onto the nodes. It has the information regarding resources available on the members of the cluster, as well as the ones required for the configured service to run and hence is able to decide where to deploy a specific service.
  - **Controller-manager:** in charge of coordination between nodes, health status checking and replication.
  - It offers the possibility to run different kinds of controllers inside the master node. A controller uses the API server to watch the shared state of the cluster and makes corrective changes to the current state to change it to the desired one.

- **Worker node:** The pods are run here, so the worker node contains all the necessary services to manage the networking between the containers, communicate with the master node, and assign resources to the containers scheduled:
  - **Docker:** runs on each of the worker nodes and runs the configured pods. It takes care of downloading the images and starting the containers.
  - **Kubelet:** gets the configuration of a pod from the apiserver and ensures that the described containers are up and running. This is the worker service that is responsible for communicating with the master node. It also communicates with etcd, to get information about services and write the details about newly created ones.
  - **kube-proxy:** acts as a network proxy and a load balancer for a service on a single worker node. It takes care of the network routing for TCP and UDP packets.
  - **Kubectl:** a command line tool to communicate with the API service and send commands to the master node.

The main features of Kubernetes are:

- Replication of components
- Auto-scaling
- Load balancing
- Rolling updates
- Logging across components
- Monitoring and health checking
- Service discovery
- Authentication

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** RedHat (CoreOS) and Cloud Native Computing Foundation	**License:** Apache 2.0

6.3.2 Mesos

Presentation

Mesos (http://mesos.apache.org/) abstracts CPU, memory, storage, and other compute resources away from machines (physical or virtual), enabling fault-tolerant and elastic distributed systems to easily be built and run effectively. The Mesos kernel runs on every machine and provides applications (e.g. Hadoop, Spark, Kafka, Elasticsearch) with APIs for resource management and scheduling across entire datacentre and cloud environments.

Main concepts and features

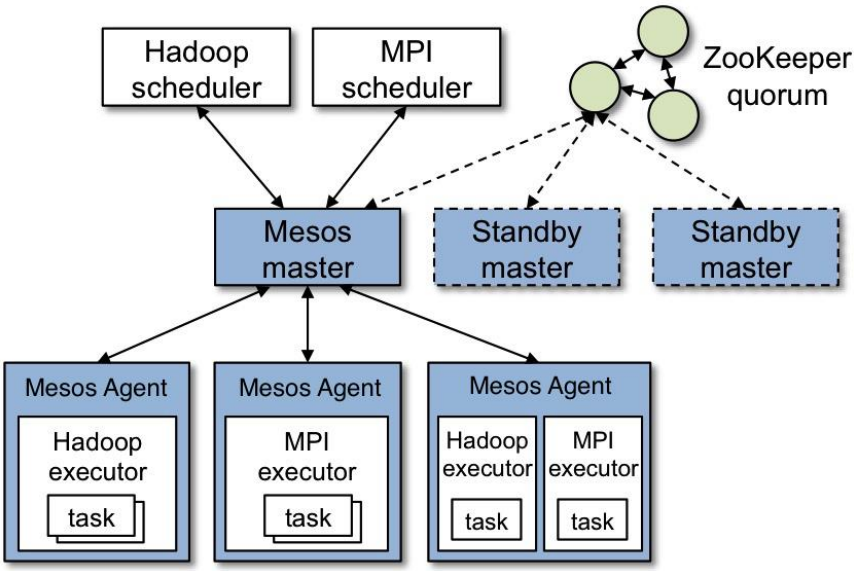

Figure 13: Mesos Architecture

A Mesos cluster is made up of four major components:

- **ZooKeepers:** a centralized configuration manager, used by distributed applications such as Mesos to coordinate activity across a cluster. Mesos uses ZooKeeper to elect a leading master and for slaves to join the cluster.
- **Mesos masters:** A Mesos master is a Mesos instance in control of the cluster. A cluster will typically have multiple Mesos masters to provide fault-tolerance, with one instance elected the leading master.
- **Mesos slave:** is a Mesos instance which offers resources to the cluster. They are the 'worker' instances - tasks are allocated to the slaves by the Mesos master.
- **Frameworks:** On its own, Mesos only provides the basic "kernel" layer of your cluster. It lets other applications request resources in the cluster perform tasks but does nothing itself. Frameworks bridge the gap between the Mesos layer and your applications. They are higher level abstractions which simplify the process of launching tasks on the cluster.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Apache	**License:** Apache 2.0

6.3.3 Zookeeper

Presentation

ZooKeeper (https://zookeeper.apache.org/) is an effort to develop and maintain an open source server which enables highly reliable distributed coordination. It is essentially a distributed hierarchical key-value store, which is used to provide a distributed configuration service, synchronization service, and naming registry for large distributed systems. ZooKeeper was a sub-project of Hadoop but is now a top-level project in its own right.

Main concepts and features

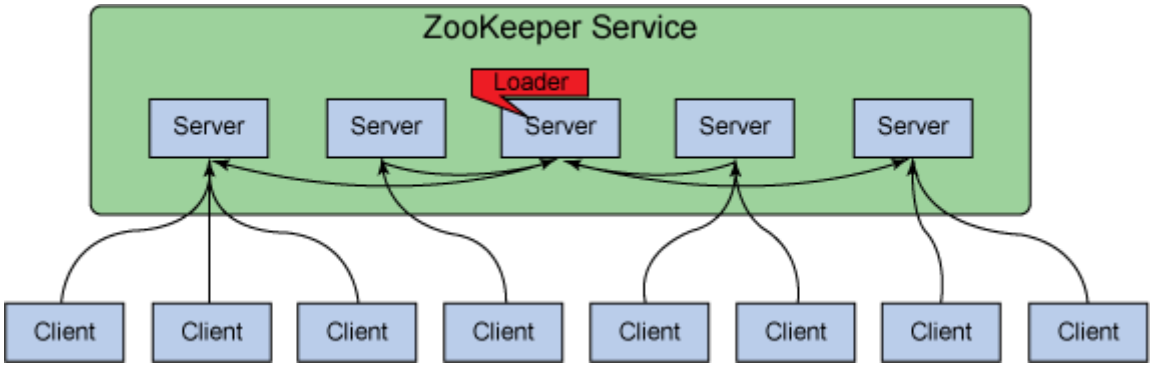

Figure 14: Zookeeper Architecture

ZooKeeper follows a simple client-server model where clients are nodes (i.e. machines) that make use of the service, and servers are nodes that provide the service. A collection of ZooKeeper servers forms a ZooKeeper ensemble. At any given time, one ZooKeeper client is connected to one ZooKeeper server. Each ZooKeeper server can handle a large number of client connections at the same time. Each client periodically sends pings to the ZooKeeper server it is connected to let it know that it is alive and connected. The ZooKeeper server in question responds with an acknowledgment of the ping, indicating the server is alive as well. When the client does not receive an acknowledgment from the server within the specified time, the client connects to another server in the ensemble, and the client session is transparently transferred over to the new ZooKeeper server.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Apache	**License:** Apache 2.0

6.3.4 Docker Swarm

Presentation

Docker Swarm (https://docs.docker.com/engine/swarm/) is a native clustering system for Docker. It turns a pool of Docker hosts into a single, virtual host using an API proxy system. It is Docker's first container orchestration project that began in 2014. Combined with Docker Compose, it is a very convenient tool to schedule containers. Its flexibility and simplicity make it easy to integrate with existing IT infrastructure.

Main concepts and features

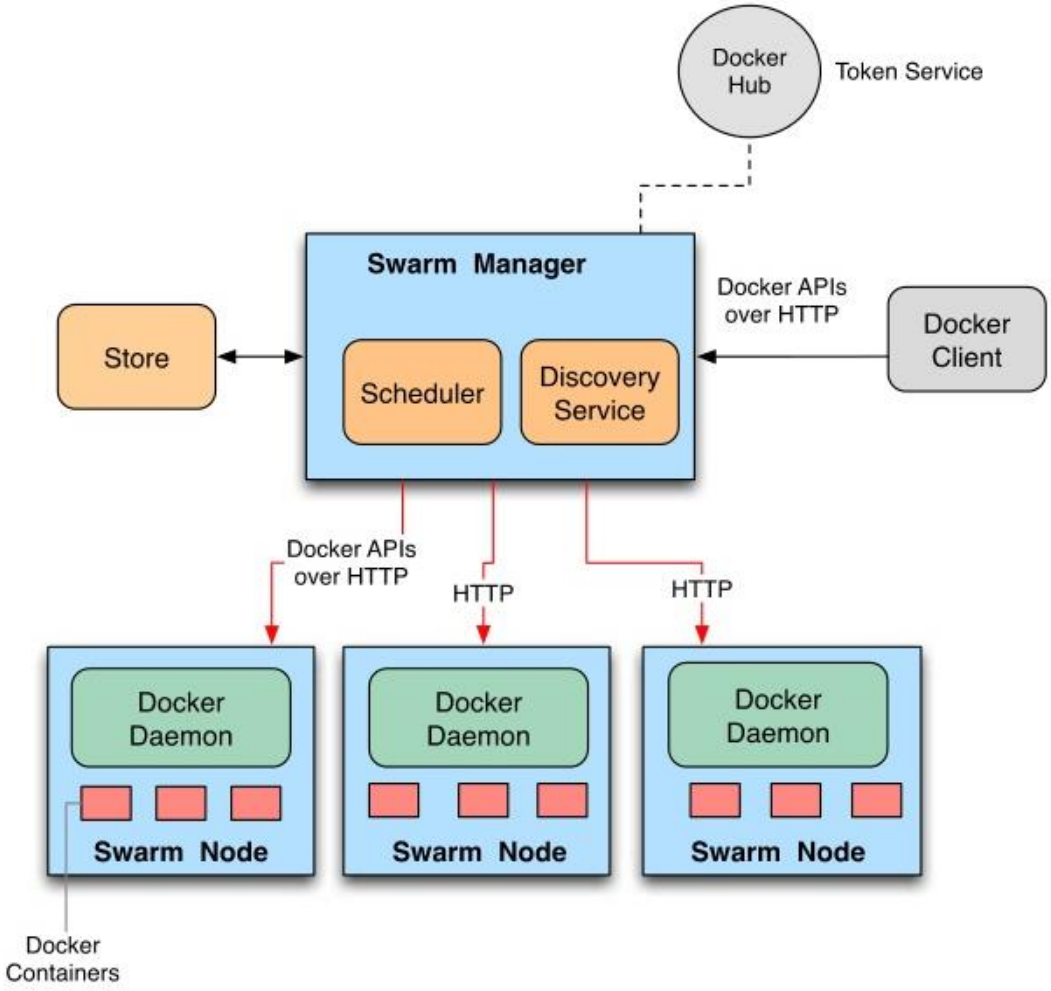

Figure 15: Docker Swarm Architecture

A swarm consists of multiple Docker hosts which run in swarm mode and act as managers (to manage membership and delegation) and workers (which run swarm services). A given Docker host can be a manager, a worker, or perform both roles. When creating a service, its optimal state (number of replicas, network and storage resources available to it, ports the service exposes to the outside world, and more) is defined. Docker works to maintain that desired state. For instance, if a worker node becomes unavailable, Docker schedules that node's tasks on other nodes. A task is a running container which is part of a swarm service and managed by a swarm manager, as opposed to a standalone container.

Docker Swarm key features are:

- Cluster management integrated with Docker Engine: Use the Docker Engine CLI to create a swarm of Docker Engines where application services can be deployed. No additional orchestration software to create or manage a swarm are needed.
- Decentralized design: Instead of handling differentiation between node roles at deployment time, the Docker Engine handles any specialization at runtime. Both kinds of nodes, managers and workers, using the Docker Engine can be deployed which means an entire swarm can be built from a single disk image.
- Declarative service model: Docker Engine uses a declarative approach to let users define the desired state of the various services in an application stack.
- Scaling: For each service, the number of tasks users want to run is declared. When scaling up or down, the swarm manager automatically adapts by adding or removing tasks to maintain the desired state.

- Desired state reconciliation: The swarm manager node constantly monitors the cluster state and reconciles any differences between the actual state and expressed desired state. For example, if a service is configured to run 10 replicas of a container, and a worker machine hosting two of those replicas crashes, the manager creates two new replicas to replace the replicas that crashed. The swarm manager assigns the new replicas to workers that are running and available.
- Multi-host networking: an overlay network for services can be specified. The swarm manager automatically assigns addresses to the containers on the overlay network when it initializes or updates the application.
- Service discovery: Swarm manager nodes assign each service in the swarm a unique DNS name and load balances running containers.
- Load balancing: Ports for services to an external load balancer can be exposed. Internally, the swarm lets specify how to distribute service containers between nodes.
- Secure by default: Each node in the swarm enforces TLS mutual authentication and encryption to secure communications between itself and all other nodes. The option to use self-signed root certificates or certificates from a custom root CA is available.
- Rolling updates: The swarm manager controls the delay between service deployment to different sets of nodes. If anything goes wrong, a task is rolled back to a previous version of the service.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Docker	**License:** Apache 2.0

6.3.5 Yarn

Presentation

A part of the Hadoop project, Yarn (https://yarnpkg.com/lang/en/) splits up the functionalities of resource management and job scheduling/monitoring into separate daemons. The idea is to have a global Resource Manager (RM) and per-application Application Master (AM). An application is either a single job or a DAG of jobs.

Architecture and main concepts

As illustrated in Figure 16, a Yarn architecture is composed of:

- The **ResourceManager** and the **NodeManager** form the data-computation framework. The ResourceManager is the ultimate authority that arbitrates resources among all the applications in the system. The NodeManager is the per-machine framework agent who is responsible for containers, monitoring their resource usage (cpu, memory, disk, network) and reporting the same to the ResourceManager/Scheduler.
- The per-application **ApplicationMaster** is, in effect, a framework specific library and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the tasks.
- The ResourceManager has two main components: Scheduler and ApplicationsManager.
- The **Scheduler** is responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues etc. The Scheduler is pure scheduler in the sense that it performs no monitoring or tracking of status for the application. Also, it offers no guarantees about restarting failed tasks either due to application failure or hardware failures. The Scheduler performs its scheduling function based on the resource requirements of the applications; it does so based on the abstract notion of a resource Container which incorporates elements such as memory, cpu, disk, network, etc.
- The Scheduler has a pluggable policy which is responsible for partitioning the cluster resources among the various queues, applications etc. The current schedulers such as the CapacityScheduler and the FairScheduler would be some examples of plug-ins.

- The **ApplicationsManager** is responsible for accepting job-submissions, negotiating the first container for executing the application specific ApplicationMaster and provides the service for restarting the ApplicationMaster container on failure. The per-application ApplicationMaster has the responsibility of negotiating appropriate resource containers from the Scheduler, tracking their status and monitoring for progress.

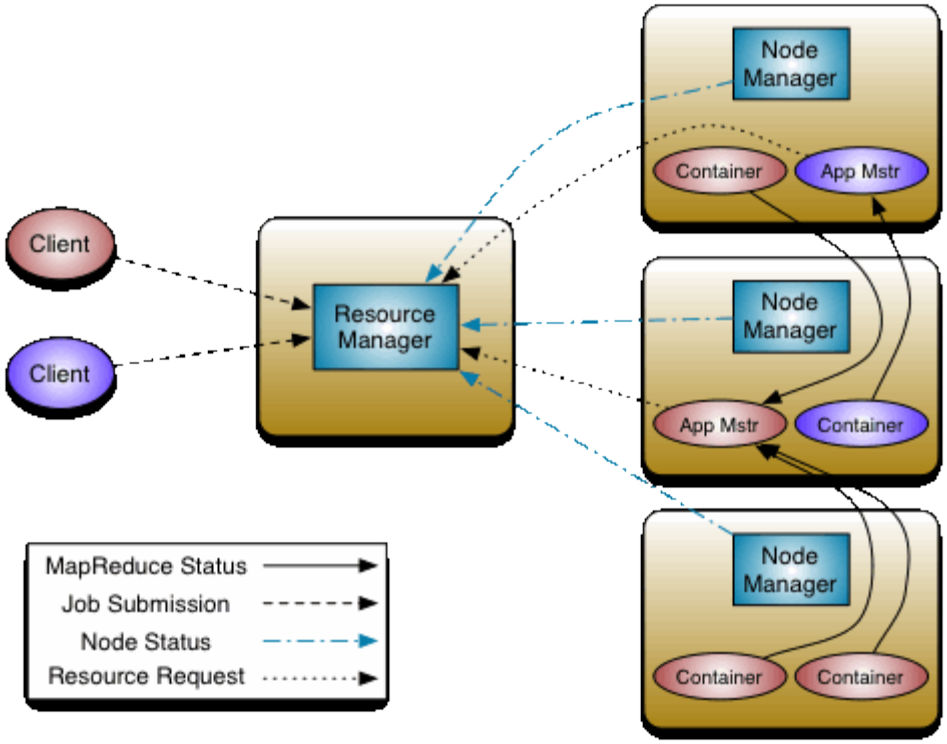

Figure 16: Yarn Architecture

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Apache	**License:** Apache 2.0

6.3.6 Comparison of Orchestration Software

Table 2: Comparison of Orchestration Software

Criteria	Kubernetes	Mesos	Zookeeper	Swarm	Yarn
Application definition	Applications can be deployed as pods.	Applications use Mesos frameworks to get resources.	Application are clients managed by zookeeper servers.	Application can be deployed as services.	Applications are deployed as nodes. An ApplicationMaster is defined for each applications.
Application scalability	Th Scaling can be manual or automated.	Due to non-monolithic scheduler, Mesos is highly scalable.	Replication is used for scalability.	Services can only be scaled manually.	Because of its monolithic scheduler, it is less scalable.
High Availability	Pods are distributed among pods to provide high availability.	Master agents are replicated among the cluster using Zookeeper.	Zookeeper servers form a cluster with one leader and followers. if a leader dies the other remaining nodes will elect a leader among themselves.	Services are replicated among Swarm nodes.	If a resource manager fails, it recovers from its own failure by restoring its state from a persistent store. Node manager failures are managed by the resource manager.
Load balancing	Pods are exposed through a service, traffic will be load balanced among them.	DC/OS, distributed operating system based on Mesos, enables service discovery and load balancing.	It offers built-in load balancing capability.	A DNS component is responsible of load balancing incoming requests to a service name.	Load shedding to dynamically re-balance the load between NMs.
Performance and reliability	Fast pod startup time maintained for large applications.	Masters and agents node failover time-to-completion is in the order of ms.	It handles a high throughput of read-write requests in with high ratio.	Fast pod startup time maintained for large applications.	Yarn is tuned to offer a better performance (especially for resource manager recovery).

6.4 Common Services

6.4.1 Data Collection

6.4.1.1 Fluentd

Presentation

Fluentd (https://www.fluentd.org) is an open source data collector that instantly enables to have a 'Log Everything' architecture with 600 + types of systems. Fluentd treats logs as JSON. It is written primarily in C with a thin-Ruby wrapper that gives users flexibility. Fluentd's scalability has been proven in the field: its largest user currently collects logs from 500 000 + servers. Fluentd can be tested in different use cases, namely: data search, data filtering and alerting, data analytics, data collection, data archiving.

Main concepts and features

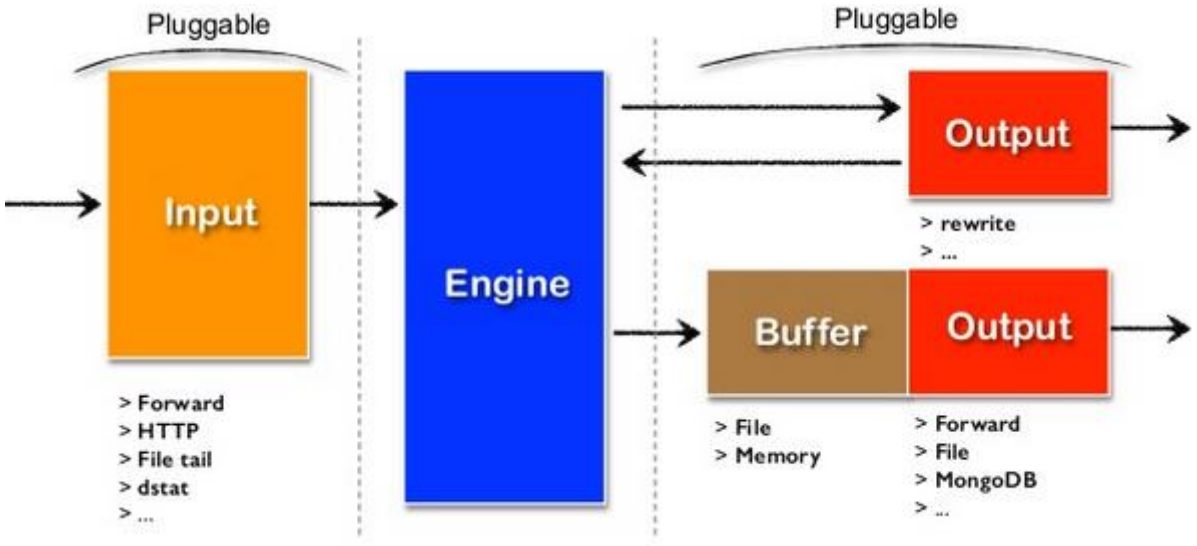

Figure 17: Fluentd Architecture

As illustrated in Figure 17, Fluentd has a flexible plugin architecture that allows the community to extend its functionality. Fluentd has 6 types of plugins: Input, Parser, Filter, Output, Formatter and Buffer.

- **Input** plugin: it extends Fluentd to retrieve and to pull event logs from external sources. An input plugin typically creates a thread socket and a listen socket. It can also be written to periodically pull data from data sources. The list of Fluentd' input plugin is:
  - in_forward: it listens to a TCP socket to receive the event stream. It also listens to an UDP socket to receive heartbeat messages;
  - in_unix: it enables Fluentd to retrieve records from the Unix Domain Socket;
  - in_http: which enables Fluentd to retrieve records from HTTP POST. The URL path becomes the tag of the Fluentd event log and the POSTed body element becomes the record itself;
  - in_tail: it allows Fluentd to read events from the tail of text files. Its behaviour is similar to the tail -F command;
  - in_exec: it executes external programs to receive or pull event logs. It will then read TSV (tab separated values), JSON or MessagePack from the stdout of the program;
  - in_syslog: enables Fluentd to retrieve records via the syslog protocol on UDP or TCP.
- **Parser** plugin: it is used when the format parameter for input plugins (exp. in_tail, in_syslog, in_tcp and in_udp) cannot parse the user's custom data format (for example, a context-dependent grammar that cannot be parsed with a regular expression) to enable the user to create their own parser formats. The list of fluentd's parser plugin is: Regexp, apache2, apache_error, nginx, syslog, csv, tsv, ltsv, json, multiline, none. The Core Input Plugins with Parser support are: in_tail, in_tcp, in_udp, in_syslog, and in_http.
- Filter plugin: it enables Fluentd to modify event streams. Example of Filter plugin' use cases are: filtering out events by grepping the value of one or more fields, enriching events by adding new fields, and deleting or masking certain fields for privacy and compliance. The list of filter plugin are: grep, record-transformer, and filter_stdout.
- **Output** plugin: There are three types of output plugins:
  - Non-Buffered output plugins do not buffer data and immediately write out results. The Non-Buffered output plugins are: out copy, out_null, out_roundrobin and out-stdout.

- Buffered output plugins maintain a queue of chunks (a chunk is a collection of events), and its behaviour can be tuned by the "chunk limit" and "queue limit" parameters (see figure 18). The Buffered output plugins are: out_exec_filter, out_forward, out_mongo.
- Time Sliced output plugins are in fact a type of Bufferred plugin, but the chunks are keyed by time (see figure 18). The Time Sliced output plugins are: out_exec, out-file, out_s3, out_webhdfs.

- **Formatter** plugin: It is used when the output format for an output plugin does not meet user's needs to let him extending and re-using custom output formats. The list of formatter plugin is: out_file, json, ltsv, csv, msgpack, hash, single_value.
- **Buffer** plugin: it is used by output plugins. For example, out_s3 uses buf_file plugin by default to store incoming stream temporally before transmitting to S3. A buffer is essentially a set of "chunks". A chunk is a collection of records concatenated into a single blob. Chunks are periodically flushed to the output queue and then sent to the specified destination. The Fluentd' Buffer available plugins are: buf_memory, and buf_file.

Fluentd key features are:

- Unified Logging with JSON: it tries to structure data as JSON as much as possible: this allows Fluentd to unify all facets of processing log data: collecting, filtering, buffering, and outputting logs across multiple sources and destinations (Unified Logging Layer).
- Pluggable architecture: it has a flexible plugin system that allows the community to extend its functionality.
- Minimum Resources required: it is written in a combination of C language and Ruby, and requires very little system resource. The vanilla instance runs on 30 - 40 MB of memory and can process 13 000 events/second/core.
- Built-in reliability: it supports memory- and file-based buffering to prevent inter-node data loss. Fluentd also support robust failover and can be set up for high availability.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Cloud Native Computing Foundation	**License:** Apache 2.0

6.4.1.2 Logstash

Presentation

Logstash (https://www.elastic.co/products/logstash) is an open source, server-side data processing pipeline that ingests data from a multitude of sources simultaneously, transforms it, and then sends it to elastic search. It supports a variety of inputs that pull in events from a multitude of common sources, all at the same time. Easily ingest from logs, metrics, web applications, data stores, and various AWS services, all in continuous, streaming fashion.

Main concepts and features

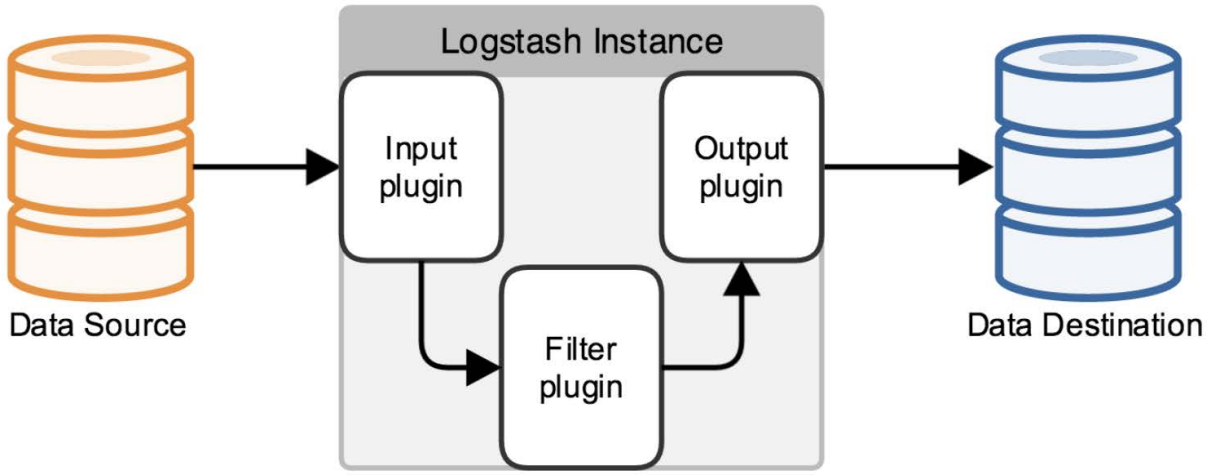

Figure 18: Logstash Architecture

Logstash can filters each event, identify named fields to build structure, and transforms them to converge on a common format for easier, accelerated analysis and business value.

Some Logstash key features differentiate it from other management tools:

- Seamless integration with Elasticsearch and Kibana: This is a powerful data processing pipeline which is tightly coupled to fetch data from multiple systems, store the data for real time search capabilities in Elasticsearch and then visualize the stored data with Kibana.
- Extensibility: It provides a variety of inputs, filters and outputs to use for processing logs of several types. It provides flexibility to create and develop inputs, filters and outputs.
- Interoperability: It provides the interoperability to use it with various other components and tools.
- Pluggable data pipeline: Il contains over 200 plugins developed by Elastic which can be used.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Elasticsearch	**License:** Apache 2.0

6.4.1.3 Beats

Presentation

Beats (https://www.elastic.co/products/beats) is an elastic platform offering data shippers which are installed as lightweight agents and send data from thousands of machines to Logstash or Elasticsearch. The Beats Family is composed of:

- **Filebeat:** offers a lightweight way to forward and centralize log and files.
- **Metricbeat:** sends system and service statistics: CPU usage, memory, file system, disk IO, and network IO metrics.
- **Packetbeat:** offers a lightweight way to analyse Network traffic in real time with zero latency overhead and without interfering with the infrastructure.
- **Winlogbeat:** reads any windows event log and streams it to Elasticsearch and Logstach.

- **Auditbeat:** ships Linux audit framework data and files 'integrity in real time to the rest of the elastic stack for further analysis.
- **Heartbeat:** monitors services for their availability with active probing.

Main concepts and features

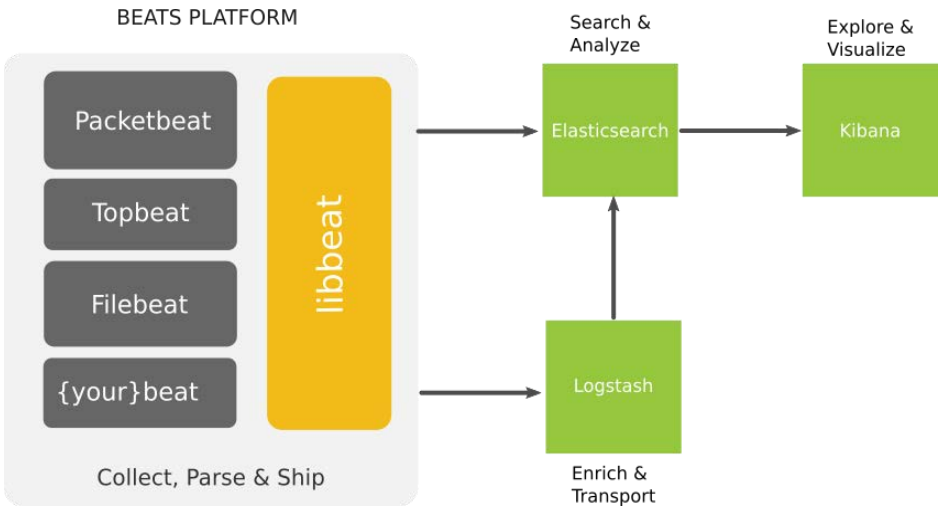

Figure 19: Beats Architecture

The Beats platform is built on a generic approach that makes easy to create new Beats. Genericity is provided by libbeat, a library that contains the common parts of all Beats for dealing with common tasks like inserting in bulk into Elasticsearch, securely sending events to Logstash, load-balancing the events to multiple Logstash and Elasticsearch nodes, and sending events in synchronous and asynchronous modes. The libbeat library also includes mechanisms for detecting when downstream servers are getting overloaded or the network in between is getting congested, so it can reduce the sending rate. The open source community works on developing new beats amongst which:

- Amazonbeat: Reads data from a specified Amazon™ product.
- Apachebeat: Reads status from Apache HTTPD server-status.
- Apexbeat: Extracts configurable contextual data and metrics from Java applications via the APEX toolkit.
- Burrowbeat: Monitors Kafka consumer lag using Burrow.
- Cassandrabeat: Uses Cassandra's nodetool cfstats utility to monitor Cassandra database nodes and lag.
- Cloudfrontbeat: Reads log events from Amazon Web Services CloudFront.
- Cloudtrailbeat: Reads events from Amazon Web Services' CloudTrail.
- Cloudwatchmetricbeat: A beat for Amazon Web Services' CloudWatch Metrics.
- Cloudwatchlogsbeat: Reads log events from Amazon Web Services' CloudWatch Logs.
- Collectbeat: Adds discovery on top of Filebeat and Metricbeat in environments like Kubernetes.
- Dockbeat: Reads Docker container statistics and indexes them in Elasticsearch.
- Elasticbeat: Reads status from an Elasticsearch cluster and indexes them in Elasticsearch.
- Gabeat: Collects data from Google™ Analytics Realtime API.
- Githubbeat: Easily monitors GitHub repository activity.
- Hsbeat: Reads all performance counters in Java HotSpot VM.

- Httpbeat: Polls multiple HTTP(S) endpoints and sends the data to Logstash or Elasticsearch. Supports all HTTP methods and proxies.
- Jmxproxybeat: Reads Tomcat JMX metrics exposed over JMX Proxy Servlet to HTTP.
- Kafkabeat: Reads data from Kafka topics.
- Mongobeat: Monitors MongoDB instances and can be configured to send multiple document types to Elasticsearch.
- Mqttbeat: Add messages from mqtt topics to Elasticsearch.
- Mysqlbeat: Run any query on MySQL and send results to Elasticsearch.
- Nginxbeat: Reads status from Nginx.
- Nginxupstreambeat: Reads upstream status from nginx upstream module.
- Pingbeat: Sends ICMP pings to a list of targets and stores the round-trip time (RTT) in Elasticsearch.
- Prombeat: Indexes Prometheus metrics.
- Prometheusbeat: Sends Prometheus metrics to Elasticsearch via the remote write feature.
- Redisbeat: Used for Redis monitoring.
- Rsbeat: Ships redis slow logs to elasticsearch and analyse by Kibana.
- Twitterbeat: Reads tweets for specified screen names.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Elasticsearch	**License:** Apache 2.0

6.4.1.4 Comparison of Data Collection Software

Table 3: Comparison of Data Collection Software

Criteria	Fluentd	Logstash	Beats
Event routing	Tags (declarative approach).	Algorithmic statements (procedural approach).	Tags
Data Transportation	Built-in reliability based on in-memory and in-disk buffering system.	Needs to be deployed with Redis to ensure reliability.	Built-in reliability based on in-memory and in-disk buffering system.
Extensibility	Supports extensibility via plugins.	Supports extensibility via plugins.	Supports extensibility via data shippers.
Performance	Less memory use	More memory use	Less memory use, and CPU.

6.4.2 Communication

6.4.2.1 Kafka

Presentation

Kafka (https://kafka.apache.org/) is a distributed streaming platform that follows a publish/ subscribe architecture to manage data streams. It is used for two broad classes of application:

1) Building real-time streaming data pipelines that reliably get data between systems or applications; and
2) Building real-time streaming applications that transform or react to the streams of data.

Kafka runs as a cluster on one or more servers. The Kafka cluster stores streams of records in categories called topics.

Main concepts and features

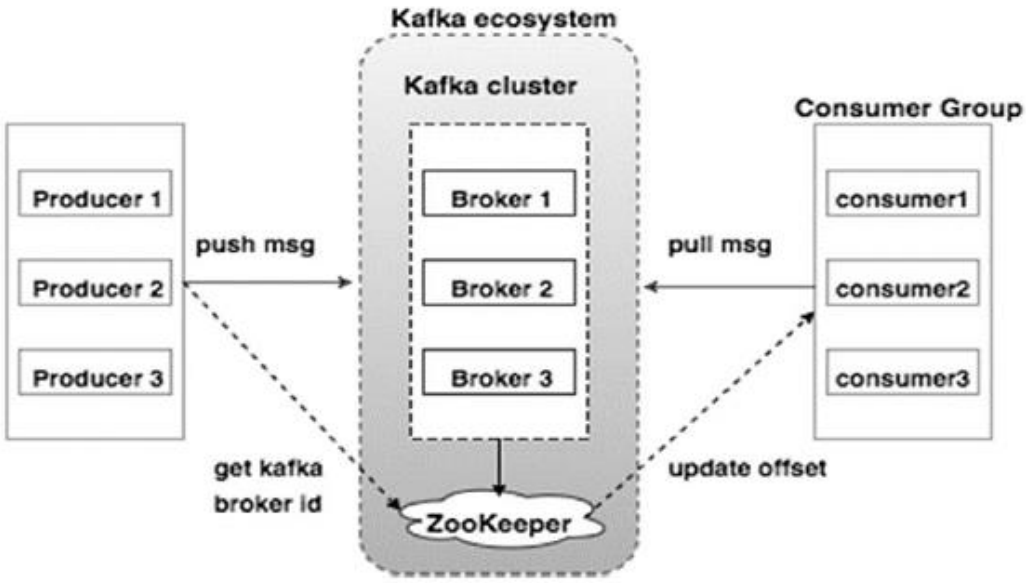

Figure 20: Kafka Architecture

A Kafka cluster primarily has five main components:

- **Topic:** is a category or feed name to which messages are published by the message producers. Topics are partitioned, and each partition is represented by the ordered immutable sequence of messages. Each message in the partition is assigned a unique sequential ID called offset.
- **Broker:** A Kafka cluster consists of one or more servers where each one may have one or more server processes running and is called the broker. Topics are created within the context of broker processes.
- **Producers:** publish data to the topics by choosing the appropriate partition within the topic.
- **Consumers:** are the applications or processes that subscribe to topics and process the feed of published messages.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Apache	**License:** Apache 2.0

6.4.2.2 Amazon™ Kinesis

Presentation

Amazon™ Kinesis (https://aws.amazon.com/kinesis/?nc1=h_ls) is a platform for streaming data on AWS (Amazon™ Web Services), offering powerful services to make it easy to load and analyse streaming data, and also providing the ability to build custom streaming data applications for specialized needs.

Main concepts and features

Amazon™ Kenesis platform encloses the following products:

- **Kinesis Video Streams** to capture, process, and store video streams for analytics and machine learning.

- **Kinesis Data Streams** to build custom applications that analyse data streams using popular stream processing frameworks.
- **Kinesis Data Firehose** to load data streams into AWS data stores.
- **Kinesis Data Analytics** to analyse data streams with SQL.

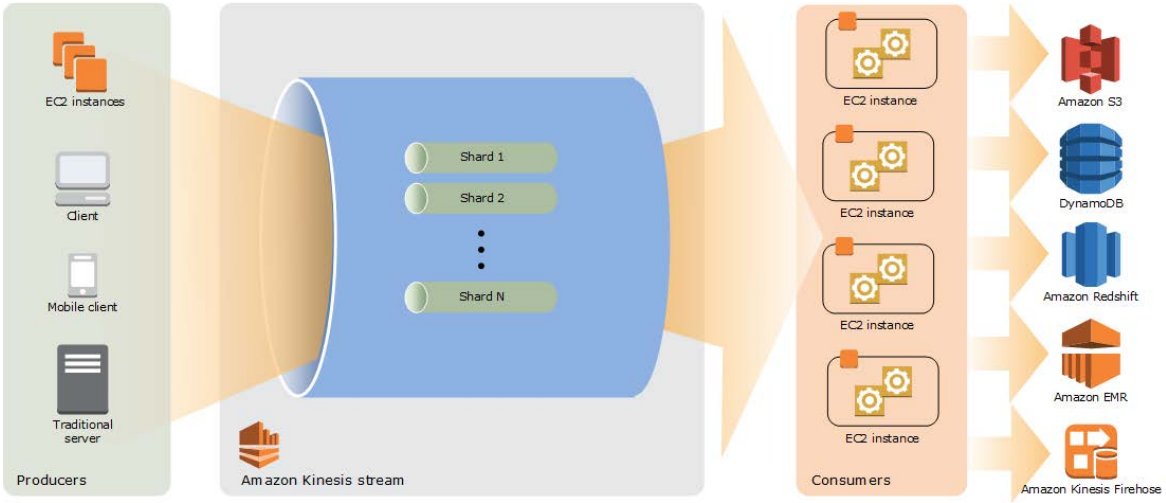

Figure 21: Amazon Kinesis High-Level Architecture

As illustrated in Figure 21, producers continually push data to Kinesis Data Streams and the consumers process the data in real time. Consumers (such as a custom application running on Amazon™ EC2, or an Amazon™ Kinesis Data Firehose delivery stream) can store their results using an AWS service such as Amazon™ DynamoDB, Amazon™ Redshift, or Amazon™ S3.

Amazon™ Kinesis key features are:

- Connect and stream from millions of devices.
- Durably store, encrypt and index data.
- fault-tolerant consumption of data from streams and scaling support for Amazon Kinesis Data Streams applications.
- Data monitoring based on real-time and customized metrics.
- Focus on managing applications instead of infrastructure.
- Build real-time and batch applications on data streams.
- Stream data in a secured way.

More detailed descriptions of some components of Amazon™ Kenesis are provided below.

- Amazon™ Kinesis Video Streams is a fully managed AWS service that streams live video from devices (smartphones, security cameras, webcams, cameras embedded in cars, drones, etc.) to the AWS Cloud, or build applications for real-time video processing or batch-oriented video analytics. It also manages non-video time-serialized data such as audio data, thermal imagery, depth data, radar data, and more. Data can be accessed frame-by-frame, in real time for low-latency processing.

Kinesis video stream can be configured to durably store media data for the specified retention period. In addition, it time-indexes stored data based on both the producer time stamps and ingestion time stamps. Applications can run on Amazon™ EC2 instances. These applications might process data using open source deep-learning algorithms or use third-party applications that integrate with Kinesis Video Streams. In the other hand, Amazon™ Kinesis Data Streams is used to collect and process large streams of data records in real time. A typical Amazon™ Kinesis Data Streams application reads data from a Kinesis data stream as data records. Applications can use the Kinesis Client Library, and they can run on Amazon™ EC2 instances. The processed records can be sent to dashboards, used to generate alerts, dynamically change pricing and advertising strategies, or send data to a variety of other AWS services. Kinesis Data Streams is part of the Kinesis streaming data platform, along with Amazon™ Kinesis Data Firehose.

- Amazon™ Kinesis Data Firehose is a fully managed service for delivering real-time streaming data to destinations such as Amazon™ Simple Storage Service (Amazon™ S3), Amazon Redshift, Amazon™ Elasticsearch Service (Amazon™ ES), and Splunk. Kinesis Data Firehose is part of the Kinesis streaming data platform, along with Kinesis Streams and Amazon™ Kinesis Data Analytics. Based on Kinesis Data Firehose, developers do not need to write applications or manage resources, they have only to configure producers to send data to Kinesis Data Firehose, and it automatically delivers the data to the specified destination.

- With Amazon™ Kinesis Data Analytics, streaming data can be processed and analysed using SQL. It can be configured to send the results to desired destinations. Kinesis Data Analytics supports Amazon™ Kinesis Data Firehose (Amazon™ S3, Amazon™ Redshift, and Amazon™ Elasticsearch Service), AWS Lambda, and Amazon™ Kinesis Data Streams as destinations.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** AWS	**License:** Commercial (with some open source components)

6.4.2.3 Flume

Presentation

Apache Flume (https://flume.apache.org/) is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving substantial amounts of log data. It has a simple and flexible architecture based on streaming data flows. It is robust and fault tolerant with tunable reliability mechanisms and many failover and recovery mechanisms. It uses a simple extensible data model that allows for online analytic application.

Main concepts and features

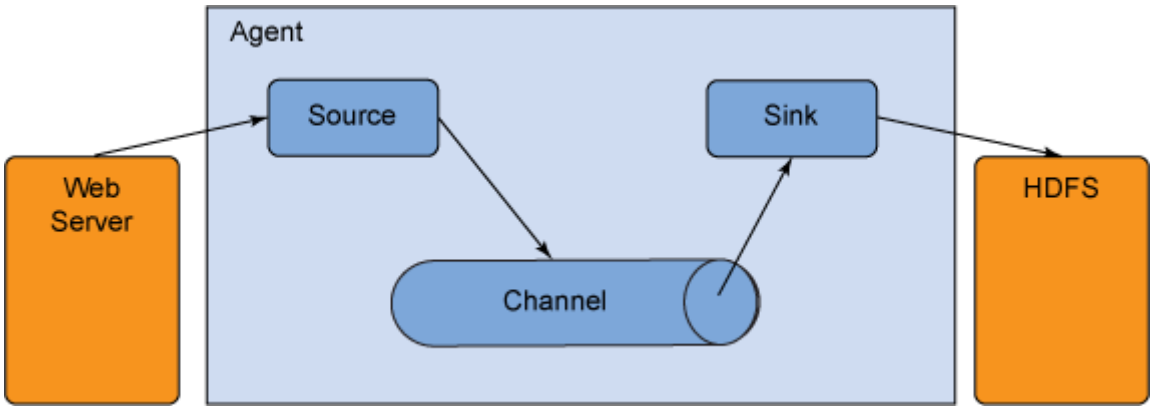

Figure 22: Flume Architecture

A Flume event is defined as a unit of data flow having a byte payload and an optional set of string attributes. A Flume agent is a (JVM) process that hosts the components through which events flow from an external source to the next destination (hop).

A Flume source consumes events delivered to it by an external source like a web server. The external source sends events to Flume in a format that is recognized by the target Flume source. For example, an Avro Flume source can be used to receive Avro events from Avro clients or other Flume agents in the flow that send events from an Avro sink. When a Flume source receives an event, it stores it into one or more channels. The channel is a passive store that keeps the event until it is consumed by a Flume sink. The sink removes the event from the channel and puts it into an external repository like HDFS (via Flume HDFS sink) or forwards it to the Flume source of the next Flume agent (next hop) in the flow. The source and sink within the given agent run asynchronously with the events staged in the channel. This Flume event flow is described in figure 22.

Flume allows a user to build multi-hop flows where events travel through multiple agents before reaching the final destination. It also allows fan-in and fan-out flows, contextual routing and backup routes (fail-over) for failed hops.

Flume main key features are:

- Reliability: single-hop message delivery semantics in Flume provide end-to-end reliability of the flow. Events are staged in a channel on each agent. The events are removed from a channel only after they are stored in the channel of next agent or in the terminal repository. Additionally, Flume uses a transactional approach to guarantee the reliable delivery of the events. The sources and sinks encapsulate in a transaction the storage/retrieval, respectively, of the events placed in or provided by a transaction provided by the channel. In the case of a multi-hop flow, the sink from the previous hop and the source from the next hop both have their transactions running to ensure that the data is safely stored in the channel of the next hop.

- Recoverability: Flume supports a durable file channel which is backed by the local file system and manages recovery from failure. There is also a memory channel which simply stores the events in an in-memory queue.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Apache	**License:** Apache 2.0

6.4.2.4 Redis

Presentation

Redis (https://redis.io/) is an open source, in-memory data structure store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries. Redis has built-in replication, Lua scripting, LRU eviction, transactions and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster.

Main concepts and features

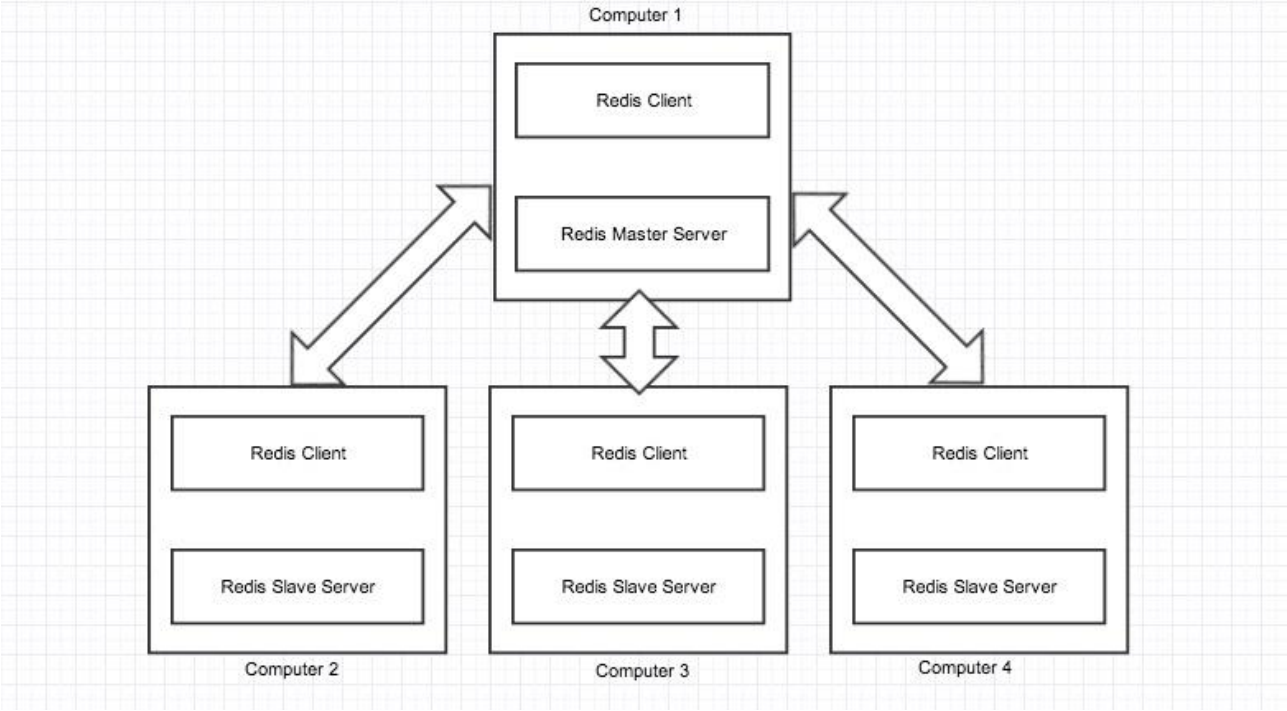

Figure 23: Redis Architecture

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Apache	**License:** Apache 2.0

6.4.2.5 Comparison of Communication Software

6.4.3 Computation

6.4.3.1 Apache Flink

Presentation

Apache Flink (https://flink.apache.org/) is an open-source stream processing framework for distributed, high-performing, always-available, and accurate data streaming applications.

Main concepts and features

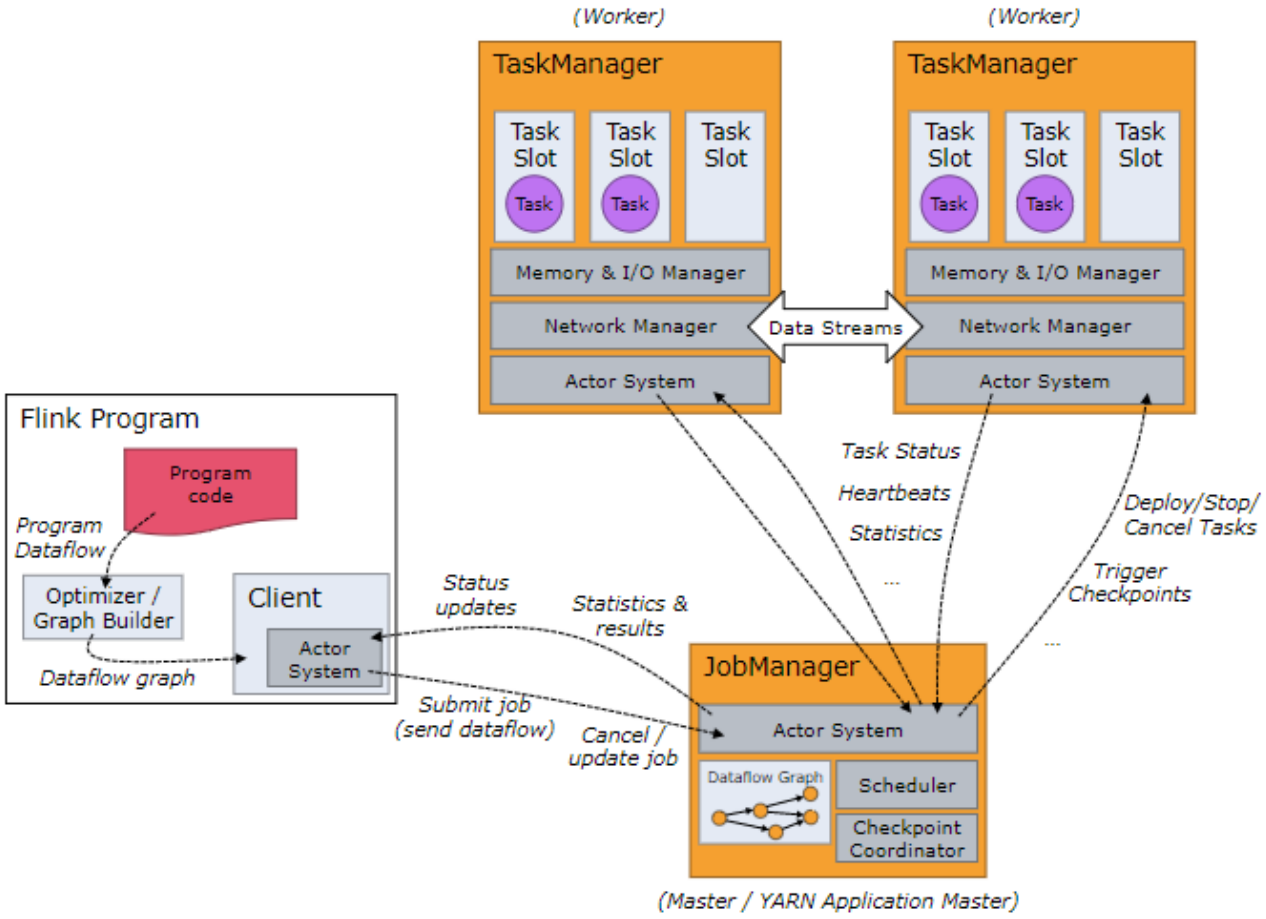

Figure 24: Flink Architecture

The Flink runtime consists of two types of processes:

- The **JobManagers** (also called masters) coordinate the distributed execution. They schedule tasks, coordinate checkpoints, coordinate recovery on failures, etc. There is always at least one Job Manager. A high-availability setup will have multiple JobManagers, one of which one is always the leader, and the others are standby.
- The **TaskManagers** (also called workers) execute the tasks (or more specifically, the subtasks) of a dataflow, and buffer and exchange the data streams. There should always be at least one TaskManager.

The JobManagers and TaskManagers can be started in various ways: directly on the machines as a standalone cluster, in containers, or managed by resource frameworks like YARN or Mesos. TaskManagers connect to JobManagers, announcing themselves as available, and are assigned work.

The **client** is not part of the runtime and program execution but is used to prepare and send a dataflow to the JobManager. After that, the client can disconnect, or stay connected to receive progress reports. The client runs either as part of the Java/Scala program that triggers the execution, or in the command line process.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Apache	**License:** Apache 2.0

6.4.3.2 Apache Spark

Presentation

Apache Spark (https://spark.apache.org/) is a fast, in-memory data processing engine with elegant and expressive development APIs to allow data workers to efficiently execute streaming, machine learning or SQL workloads that require fast iterative access to datasets.

Main concepts and features

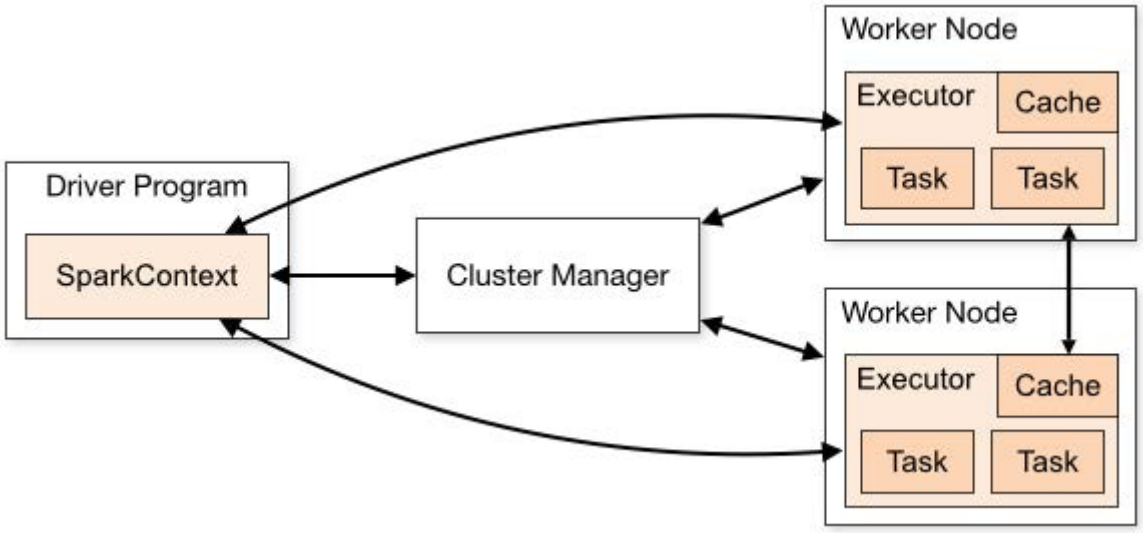

Figure 25: Spark Architecture

Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in your main program (called the driver program).

Specifically, to run on a cluster, the SparkContext can connect to several types of cluster managers which allocate resources across applications. Once connected, Spark acquires executors on nodes in the cluster, which are processes that run computations and store data for your application. Next, it sends your application code (defined by JAR or Python files passed to SparkContext) to the executors. Finally, SparkContext sends tasks to the executors to run.

- Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads. This has the benefit of isolating applications from each other, on both the scheduling side (each driver schedules its own tasks) and executor side (tasks from different applications run in different JVMs). However, it also means that data cannot be shared across different Spark applications (instances of SparkContext) without writing it to an external storage system.
- Spark is agnostic to the underlying cluster manager. As long as it can acquire executor processes, and these communicate with each other, it is relatively easy to run it even on a cluster manager that also supports other applications (e.g. Mesos/YARN).
- The driver program will listen for and accept incoming connections from its executors throughout its lifetime (e.g. see spark.driver.port in the network configuration part). As such, the driver program will be network addressable from the worker nodes.
- Because the driver schedules tasks on the cluster, it should be run close to the worker nodes, preferably on the same local area network. In order to send requests to the cluster remotely, it is better to open an RPC to the driver and use it to submit operations from nearby rather than to run a driver far away from the worker nodes.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Apache	**License:** Apache 2.0

6.4.3.3 Apache Storm

Presentation

Apache Storm (http://storm.apache.org/) is a free and open source distributed real-time computation system. Storm makes it easy to reliably process unbounded streams of data, doing for real-time processing what Hadoop did for batch processing. A Storm topology consumes streams of data and processes those streams in arbitrarily complex ways, repartitioning the streams between each stage of the computation however needed.

Main concepts and features

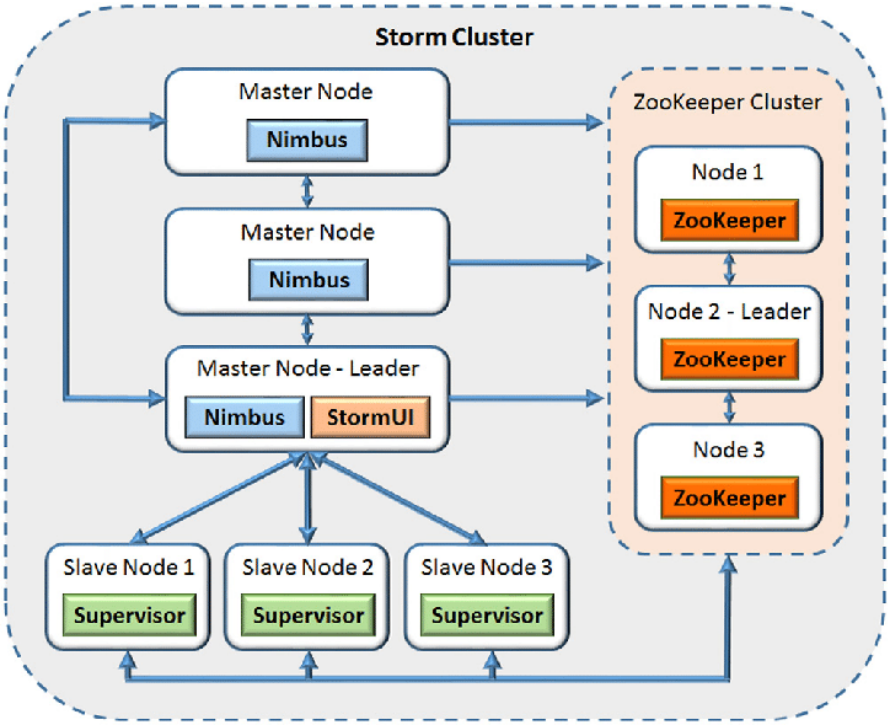

Figure 26: Storm Architecture

There are two types of nodes in Storm cluster:

- **Master node:** runs a daemon called "Nimbus", which is similar to the 'Job Tracker' of Hadoop cluster. Nimbus is responsible for distributing codes, assigning tasks to machines and monitoring their performance.
- **Slave node:** runs a daemon called "Supervisor" which is able to run one or more worker processes on its node. Each supervisor works assigned by Nimbus and starts and stops the worker processes when required. Every worker process runs a specific set of topology which consists of worker processes working around machines.

Since Apache Storm does not have the ability to manage its cluster state, it depends on Apache Zookeeper for this purpose. Zookeeper facilitates communication between Nimbus and Supervisors with the help of message acknowledgements, processing status, etc.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Apache	**License:** Apache 2.0

6.4.3.4 Apache Hadoop

Presentation

Hadoop (https://hadoop.apache.org/) is an open-source software framework for storing data and running applications on clusters of commodity hardware. It provides massive storage for any kind of data, enormous processing power and the ability to handle virtually limitless concurrent tasks or jobs.

Main concepts and features

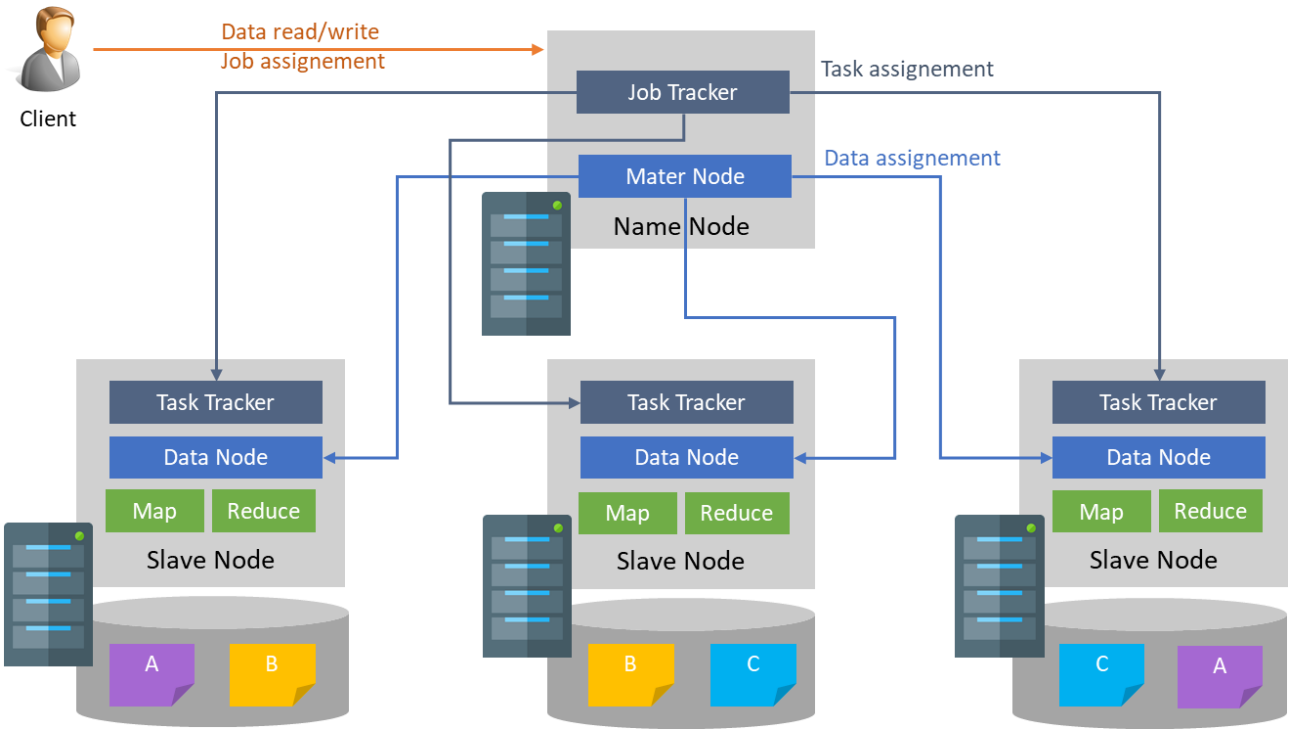

Figure 27: Hadoop Architecture

An Hadoop cluster is composed of two types of nodes: NameNode and DataNode:

- **NameNode:** keeps the directory tree of all files in the file system and tracks where across the cluster the file data is kept. It does not store the data of these files itself. Client applications talk to the NameNode whenever they wish to locate a file, or when they want to add/copy/move/delete a file. The NameNode responds the successful requests by returning a list of relevant DataNode servers where the data lives.
- **DataNode:** stores data in the HadoopFileSystem. A functional filesystem has more than one DataNode, with data replicated across them. On startup, a DataNode connects to the NameNode; spinning until that service comes up. It then responds to requests from the NameNode for filesystem operations. Client applications can talk directly to a DataNode, once the NameNode has provided the location of the data. DataNode instances can talk to each other, which is what they do when they are replicating data.

The execution of a MapReduce job begins when the client submits the job configuration to the Job Tracker that specifies the map, combine and reduce functions along with the location for input and output data. On receiving the job configuration, the job tracker identifies the number of splits based on the input path and select Task Trackers based on their network vicinity to the data sources. Job Tracker sends a request to the selected Task Trackers.

The processing of the Map phase begins where the Task Tracker extracts the input data from the splits. Map function is invoked for each record parsed by the "InputFormat" which produces key-value pairs in the memory buffer. The memory buffer is then sorted to different reducer nodes by invoking the combine function. On completion of the map task, Task Tracker notifies the Job Tracker. When all Task Trackers are done, the Job Tracker notifies the selected Task Trackers to begin the reduce phase. Task Tracker reads the region files and sorts the key-value pairs for each key. The reduce function is then invoked which collects the aggregated values into the output file.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Apache	**License:** Apache 2.0

6.4.4 Storage

6.4.4.1 Apache Cassandra

Presentation

The Apache Cassandra database(http://cassandra.apache.org/) manages fast massive amounts of data. It is scalable and high available without compromising performance. Linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure make it the perfect platform for mission-critical data. Cassandra's support for replicating across multiple datacentres is best-in- class, providing lower latency for users and some peace of mind, knowing that the system can survive regional outages.

The design goal of Cassandra is to handle big data workloads across multiple nodes without any single point of failure. Cassandra has peer-to-peer distributed system across its nodes, and data is distributed among all the nodes in a cluster:

- All the nodes in a cluster play the same role. Each node is independent and at the same time interconnected to other nodes.
- Each node in a cluster can accept read and write requests, regardless of where the data is actually located in the cluster.
- When a node goes down, read/write requests can be served from other nodes in the network.

Main concepts and features

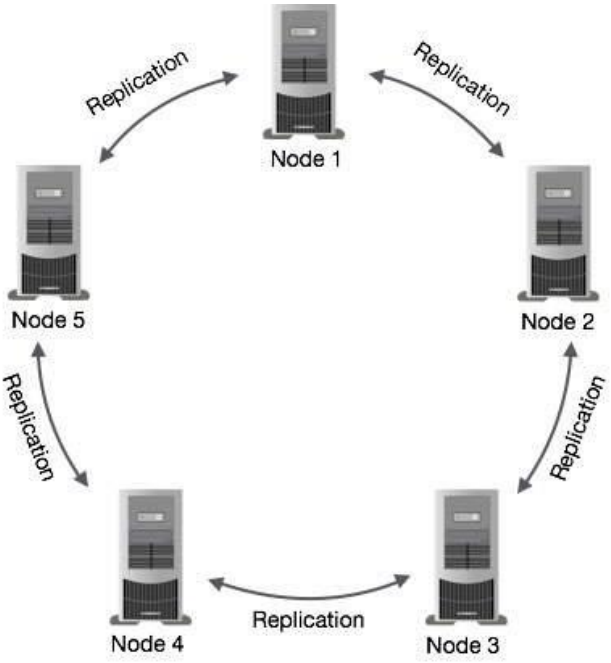

Figure 28: Cassandra Architecture

The key components of Cassandra are as follows:

- **Node:** is the place where data is stored.
- **Data centre:** is a collection of related nodes.

- **Cluster:** is a component that contains one or more data centres.
- **Commit log:** is a crash-recovery mechanism in Cassandra. Every write operation is written to the commit log.
- **Mem-table:** is a memory-resident data structure. After commit log, the data will be written to the mem-table. Sometimes, for a single-column family, there will be multiple mem-tables.
- **SSTable:** is a disk file to which the data is flushed from the mem-table when its contents reach a threshold value.
- **Bloom filter:** are quick, nondeterministic, algorithms for testing whether an element is a member of a set. It is a special kind of cache. Bloom filters are accessed after every query.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Apache	**License:** Apache 2.0

6.4.4.2 Apache Hive

Presentation

Apache Hive (https://hive.apache.org/) is an open source project. Previously a subproject of Apache Hadoop, it has now graduated to become a top-level project of its own.

The Apache Hive data warehouse software facilitates reading, writing, and managing large datasets residing in distributed storage using SQL. Structure can be projected onto data already in storage. A command line tool and JDBC driver are provided to connect users to Hive.

Main concepts and features

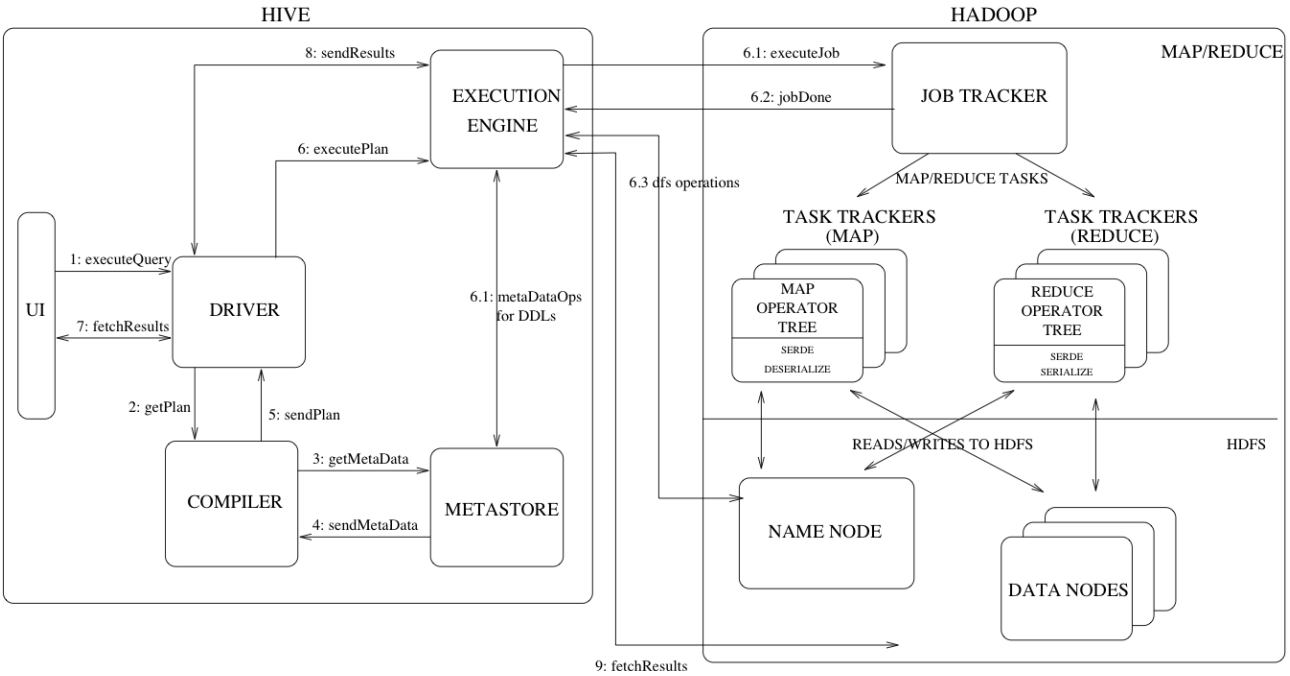

Figure 29: Apache Hive Architecture

Figure 29 shows the major components of Hive and its interactions with Hadoop. As shown in that figure, the main components of Hive are:

- **UI:** is the user interface for users to submit queries and other operations to the system. As of 2011 the system had a command line interface and a web-based GUI was being developed.

- **Driver:** is the component which receives the queries. This component implements the notion of session handles and provides execute and fetch APIs modelled on JDBC/ODBC interfaces.
- **Compiler:** is the component that parses the query, does semantic analysis on the different query blocks and query expressions and eventually generates an execution plan with the help of the table and partition metadata looked up from the metastore.
- **Metastore:** is the component that stores all the structure information of the various tables and partitions in the warehouse including column and column type information, the serializers and deserializers necessary to read and write data and the corresponding HDFS files where the data is stored.
- **Execution Engine:** is the component which executes the execution plan created by the compiler. The plan is a DAG of stages. The execution engine manages the dependencies between these various stages of the plan and executes these stages on the appropriate system components.

Hive Data Model is organized into:

- Tables: These are analogous to Tables in Relational Databases. Tables can be filtered, projected, and joined. Additionally, all the data of a table is stored in a directory in HDFS. Hive also supports the notion of external tables wherein a table can be created on pre-existing files or directories in HDFS by providing the appropriate location to the table creation DDL. The rows in a table are organized into typed columns similar to Relational Databases.
- Partitions: Each Table can have one or more partition keys which determine how the data is stored, for example a table T with a date partition column ds had files with data for a particular date stored in the <table location>/ds=<date> directory in HDFS. Partitions allow the system to prune data to be inspected based on query predicates, for example a query that is interested in rows from T that satisfy the predicate T.ds = '2008-09-01' would only have to look at files in <table location>/ds=2008-09-01/ directory in HDFS.
- Buckets: Data in each partition may in turn be divided into Buckets based on the hash of a column in the table. Each bucket is stored as a file in the partition directory. Bucketing allows the system to efficiently evaluate queries that depend on a sample of data (these are queries that use the SAMPLE clause on the table).

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Apache	**License:** Apache 2.0

6.4.4.3 Couchbase

Presentation

Main concepts and features

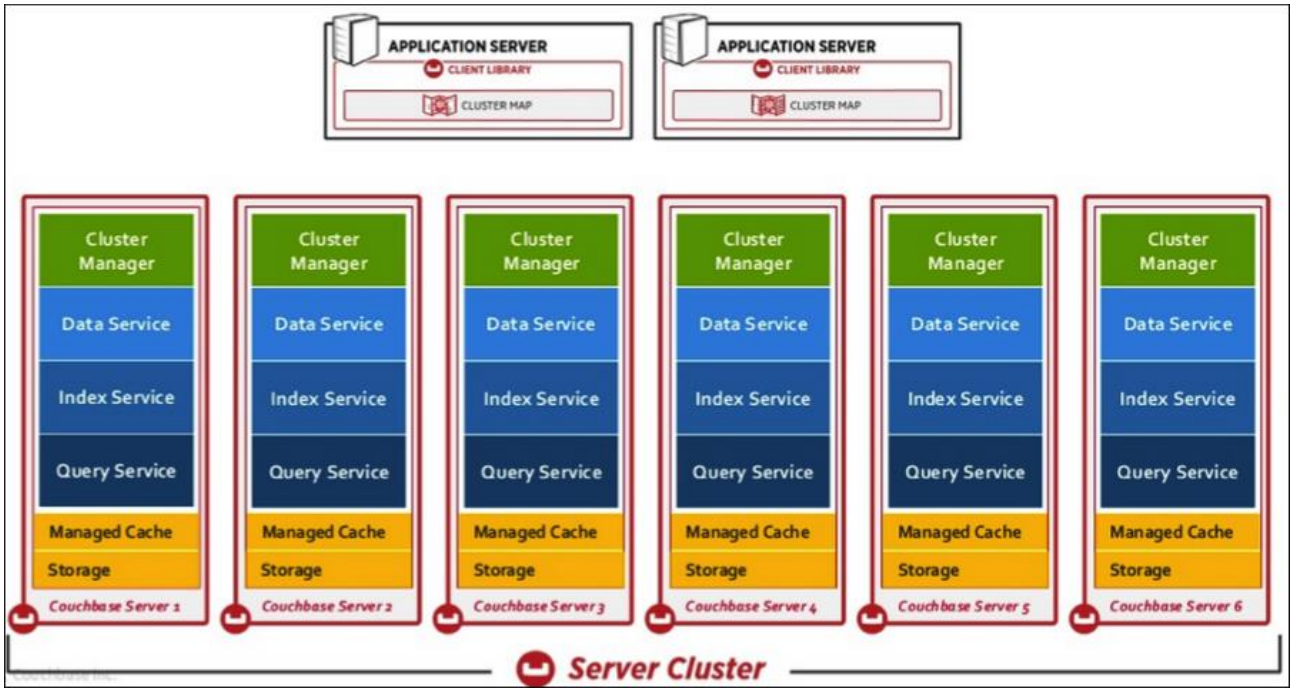

Figure 30: Couchebase Architecture

A CouchBase server is a distributed NoSQL database engine on which applications connect to perform read and write operations with low latency (< millisecond) and high throughput (millions of operations/ second).

As illustrated in Figure 30, a CouchBase Server offers services which are components that run specific workloads in the cluster. One can distinguish Data, Index, Search and Query Services to enable within a Couchbase Cluster to enable the distinct workloads such as core data operations, indexing and query processing.

The role of each CouchBase services are the following:

- **Cluster manager:** runs on all the nodes of the CouchBase cluster and orchestrations servers' operations. The cluster manager is responsible of the operations described below:
  - Cluster topology and node membership:
    - Managing node membership, adding and deleting nodes without downtime.
    - Discovery of cluster topology by internal and external connections.
    - Rebalancing loads when cluster topology changes.
    - Node health, and service monitoring.
    - Service layout across nodes.
  - Authentication.
  - Data placement:
    - Smart distribution of replicas with nodes.
    - Failure domain awareness.
  - Statistics and logging.

- **Data Service:** provides the core data access with the database engine and incremental MapReduce view processing with the views engine. Couchbase Server stores data as items. An *item* is made up of a key (also known as a document key or a document ID) and a document value, along with associated metadata. It organizes data into Buckets. Through a managed cache, a high throughput storage engine, and a memory-based replication architecture, the database engine can process a high number of concurrent requests at any moment of time at a sub-millisecond latency.

- **Index Service:** provides faster access to data in a bucket. CouchBase server supports the following indexers: Incremental Map-Reduce View indexer, Global Secondary Index (GSI) indexer, Spatial Views indexer, Full Text Search indexer. Based on these indexers, two types of indexers can be created:

  - Primary index which indexes all the keys in a bucket and is used when a full bucket scan is needed or a secondary index cannot satisfy a request.

  - Secondary index which indexes a subset of items in a given bucket.

- **Query Service:** provides separate ways to query data using either document keys, view queries, spatial queries, or N1QL queries.

The main features of Couchbase are:

- Unified Programming Interface: the Couchbase Data Platform provides simple, uniform and powerful application development APIs across multiple programming languages, connectors, and tools that makes building applications simple and accelerates time to market for applications.

- Mobile and IoT deployments: the Couchbase Data Platform provides a full-stack data platform for mobile and IoT applications, with built-in real-time data synchronization, enterprise-level security, and data integration.

- Analytics: the Couchbase Analytics provides powerful parallel query processing and is designed to efficiently execute complex, long-running queries that contain complex joins, set, aggregation, and grouping operations.

- Scale-out Architecture: Couchbase provides built-in distributed data storage and processing. This architecture ensures that the application is always on by providing the ability to detect and recover from hardware and network failures. Replication and sharding are fundamental features of a Couchbase Server. It automatically distributes data across nodes in the cluster. Thus, the database can grow horizontally to share load by adding more RAM, disk, and CPU capacity without increasing the burden on developers and administrators.

- Big Data and SQL Integration: Couchbase Data Platform includes built-in Big Data and SQL integration, thus allowing to leverage tools, processing capacity, and data wherever it may reside.

- Security.

- Container and Cloud deployments: Couchbase supports all cloud platforms, as well as a variety of container and virtualization technologies to enable operational excellence.

- High availability: Couchbase Server's built-in fault tolerance mechanisms protect against downtime caused by arbitrary unplanned incidents, including server failures. Replication and failover are important mechanisms that increase system availability. Couchbase Server replicates data across multiple nodes to support failover.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Couchbase	**License:** Apache 2.0

6.4.4.4 Apache HBase

Presentation

Apache HBase (https://hbase.apache.org/) is the Hadoop database, a distributed, scalable, big data store. The goal of the project is the hosting of very large atop clusters of commodity hardware. Apache HBase is an open-source, distributed, versioned, non-relational database. Apache HBase provides Google™ Bigtable-like capabilities on top of Hadoop and HDFS.

Main concepts and features

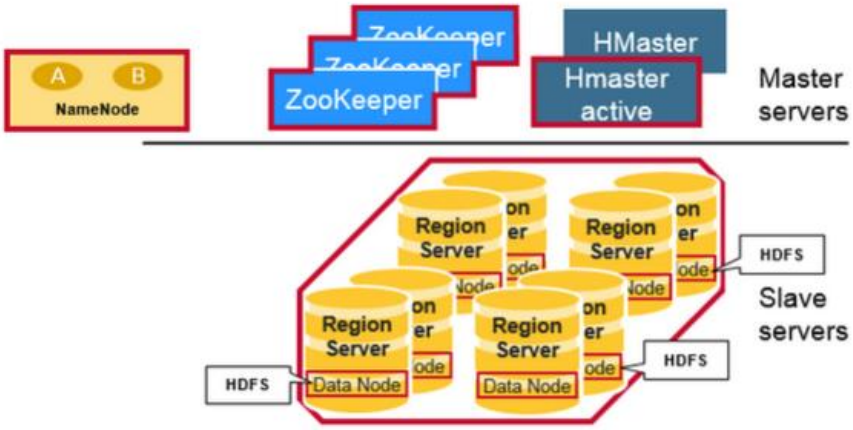

Figure 31: Apache HBase Architecture

As illustrated in Figure 31, HBase is composed of three types of servers in a master slave type of architecture which are: Region Server, HBase HMaster and Zookeeper.

- **Region servers** serve data for reads and writes. Clients communicate with HBase RegionServers directly when accessing data. The Hadoop DataNode stores the data that the Region Server is managing. All HBase data is stored in HDFS files. Region Servers are collocated with the HDFS DataNodes, which enable data locality. HBase data is local when it is written, but when a region is moved, it is not local until compaction.
- **HMaster** servers handle Region assignment, DDL (create, delete tables) operations. The NameNode maintains metadata information for all the physical data blocks that comprise the files.
- **Zookeeper**, which plays the roles of a coordinator, maintains a live cluster state.

The Hbase database's main concepts are:

- Map: Storage is in a map which is based on the key/value principle. Each value (array of bytes) is identified by a key (array of bytes). Access to a value by its key is very fast.
- Sorted Map: The map is sorted by lexicographic order. This sorting feature is very important because it allows user to retrieve values by key interval.
- Multidimensional: The key in the map is a structure composed of row-key, column family, column, and a timestamp.
- Sparse: Unlike relational databases, a column that has no value is not materialized (no storage is needed in case of a null value for a column).
- Persistence: The data stored in the map is saved permanently on disk.
- Consistency: All changes are atomic and read operations are always processed on the last committed value.
- Distributed system: The database system is built on a distributed file system so that the underlying file storage is distributed across a set of machines in a cluster. The data is replicated to many nodes thus allowing fault tolerance.

The HBase model is based on six concepts, which are:

- Table: In HBase, the data is organized in tables. A table is segregated into several partitions named Regions.
- Row: In each table the data is organized in rows. A line is identified by a unique key (RowKey). The Rowkey does not have a type, it is treated as an array of bytes.

- Column Family: Data within a row is grouped by column family. Each row of the table has the same column family, which can be populated or not. The column families are defined when the table is created in HBase.
- Column qualifier: Access to data within a column family is insured via the column qualifier or column. As RowKeys, the column qualifier is not typed, it is treated as an array of bytes.
- Cell: The combination of RowKey, Column Family and Column Qualifier uniquely identifies a cell. The data stored in a cell is called the values of that cell. Values have no type, they are always considered as byte array.
- Version: Values within a cell are versioned. The versions are identified by their timestamp. The number of versions is configured through the Column Family.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Apache	**License:** Apache 2.0

6.4.4.5 Vitess

Presentation

Vitess (https://vitess.io/) is a database solution for scaling MySQL. It is architected to run as effectively in a public or private cloud architecture as it does on dedicated hardware. It combines and extends many important MySQL features with the scalability of a NoSQL database. Vitess has been serving all YouTube™ database traffic since 2011.

Main concepts and features

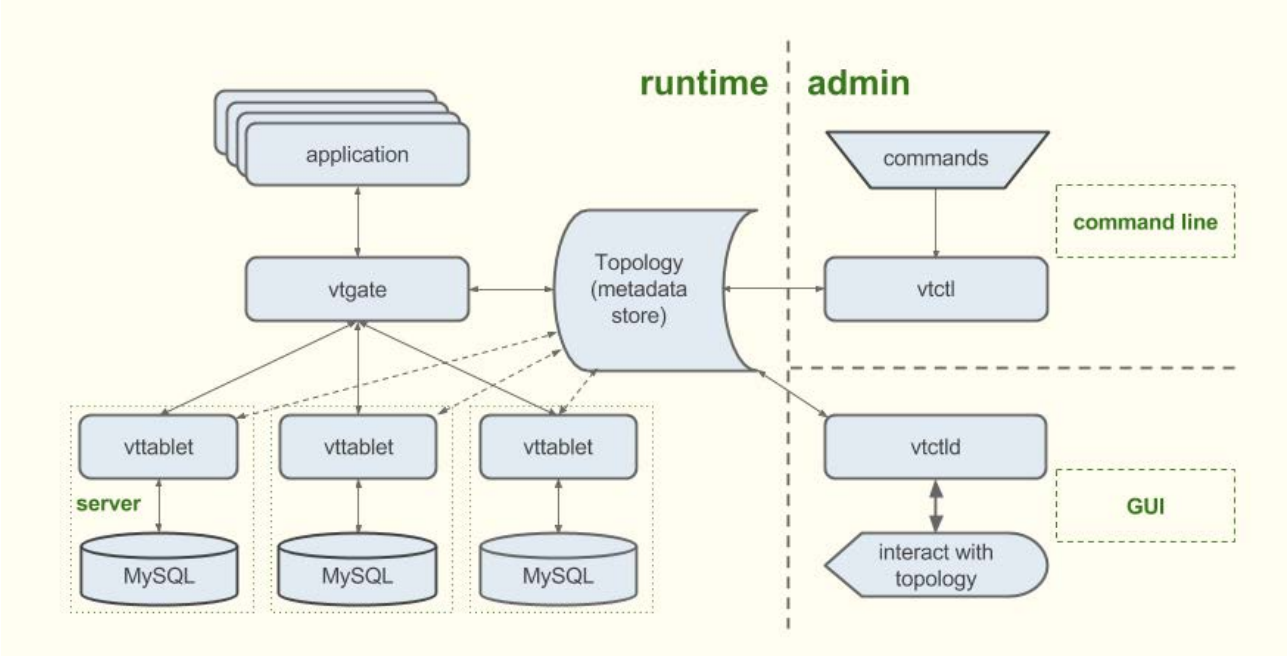

Figure 32: Vitess Architecture

The Vitess platform consists of a number of server processes, command-line utilities, and web-based utilities, backed by a consistent metadata store.

Depending on the current status of an application, one could arrive at a full Vitess implementation through a number of different process flows. For example, when building a service from scratch, the first step with Vitess would be to define the database topology. However, if the need is to scale an existing database, a likely start should be deploying a connection proxy.

Vitess tools and servers are designed to help support a complete fleet of databases as well as starting with a small database and scale over time. For smaller implementations, vttablet features like connection pooling and query rewriting help get more from the existing hardware. Vitess' automation tools then provide additional benefits for larger implementations.

Key features of Vitess are described as follows:

- Performance
  - Connection pooling: Multiplex front-end application queries onto a pool of MySQL connections to optimize performance.
  - Query de-duping: Reuse results of an in-flight query for any identical requests received while the in-flight query was still executing.
  - Transaction manager: Limit number of concurrent transactions and manage deadlines to optimize overall throughput.
- Protection
  - Query rewriting and sanitization: Add limits and avoid non-deterministic updates.
  - Query blacklisting: Customize rules to prevent potentially problematic queries from hitting your database.
  - Query killer: Terminate queries that take too long to return data.
  - Table ACLs: Specify access control lists (ACLs) for tables based on the connected user.
- Monitoring
  - Performance analysis: Tools allow to monitor, diagnose, and analyse your database performance.
  - Query streaming – Use a list of incoming queries to serve OLAP workloads.
  - Update stream – A server streams the list of rows changing in the database, which can be used as a mechanism to propagate changes to other data stores.
- Topology Management Tools
  - Master management tools (handles reparenting).
  - Web-based management GUI.
  - Designed to work in multiple data centres/regions.
- Sharding
  - Virtually seamless dynamic re-sharding.
  - Vertical and Horizontal sharding support.
  - Multiple sharding schemes, with the ability to plug-in custom ones.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Cloud Native Computing Foundation	**License:** Apache 2.0

6.4.5 Search Engine

6.4.5.1 Elasticsearch

Presentation

Elasticsearch (https://www.elastic.co/products/elasticsearch) is a distributed, RESTful search and analytics engine capable of solving a growing number of use cases. As the heart of the Elastic Stack, it centrally stores the data. Elasticsearch allows to perform and combine many types of searches - structured, unstructured, geo, metric. Elasticsearch uses standard RESTful APIs and JSON. Clients are built and maintained in many languages such as Java, Python, .NET and Groovy.

Main concepts and features

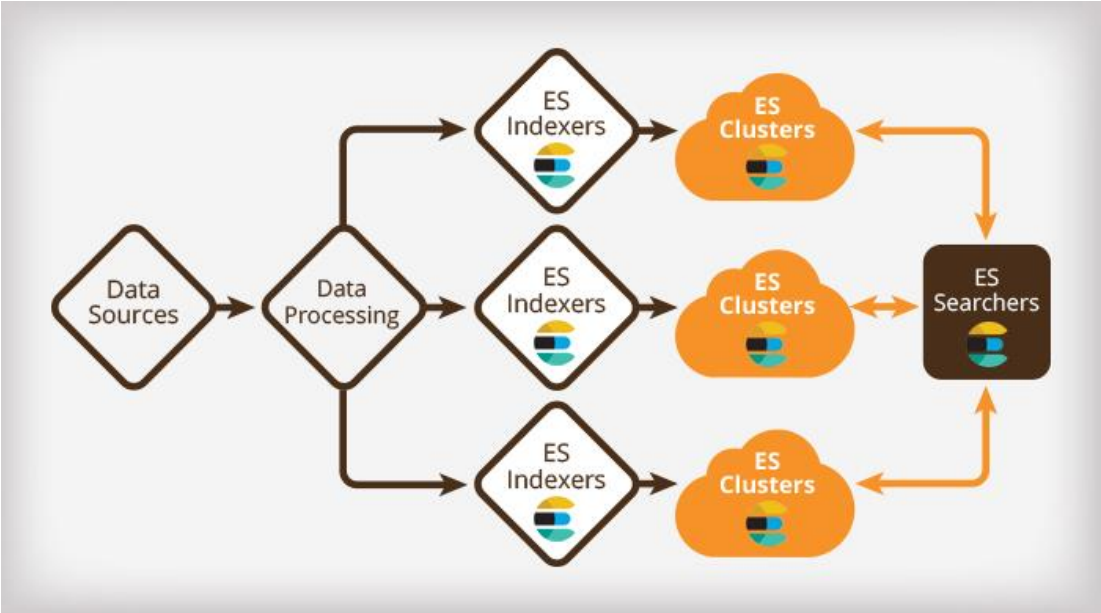

Figure 33: Elastic Search cluster

Elasticsearch can be considered as the most advanced search engine that offers:

- Real time data availability and analytics: as soon as data is indexed, it is made available for search and analytics.
- It takes JSON document, rather structured JSON documents as input to create indices. All of the field's properties are automatically detected and indexed by default. It creates mappings on its own. There is no need to define a schema. It offers full text search on data that is indexed.
- Distributed: It allows as to set up as many nodes as needed. Cluster can manage high number of nodes, and it can grow horizontally to a large number.
- Scalable: It scales horizontally to handle jillions of events per second, while automatically managing how indices and queries are distributed across the cluster for oh-so smooth operations.
- High available: the cluster is smart enough to detect a new node or a failed node to add or remove from the cluster.
- Multitenancy: An alias for index can be created. Usually a cluster contains many indices. These aliases allow a filtered view of an index to achieve multitenancy.
- Extensible: It can be extended with new features which give an enhanced experience with security, monitoring, alerting, reporting, graph exploration, and machine learning features.

"Elasticsearch dived indexes into shards and each shard can have zero or more replicas. Each node hosts one or more shards, and acts as a coordinator to delegate operations to the correct shard(s). Rebalancing and routing are done automatically". Related data is often stored in the same index, which consists of one or more primary shards, and zero or more replica shards. Once an index has been created, the number of primary shards cannot be changed.

Elasticsearch uses Lucene and tries to make all its features available through the JSON and Java API.

Elasticsearch offers a feature called "gateway" which handles the long-term persistence of the index.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Elasticsearch	**License:** Apache 2.0

6.4.5.2 Solr

Presentation

Apache Solr (http://lucene.apache.org/solr/) is an open source enterprise search platform, written in Java, from the Apache Lucene project. Providing distributed search and index replication, Solr is designed for scalability and fault tolerance. It has REST-like HTTP/XML and JSON APIs that make it usable from most popular programming languages.

Main concepts and features

Solr is a standalone enterprise search server with a REST-like API. Documents are put in it (called "indexing") via JSON, XML, CSV or binary over HTTP. Queries are sent as HTTP GET and results are received in JSON, XML, CSV or binary format.

The key features Solr are:

- Advanced Full-Text Search Capabilities: Solr provides matching capabilities including phrases, wildcards, jons, grouping and much more across any data type.
- Optimized for High Volume Traffic: Solr manages high volume of data.
- Standards-based open interfaces: Solr is using XML, JSON and HTTP to make applications building a snap.
- Easy comprehensive administration interfaces
- Easy monitoring: it publishes loads of metric data via JMX.
- High scalable and fault tolerant: Solr is built on Zookeeper which makes it easy to scale up and down. Solr bakes in replication, distribution, load balancing out of the box.
- Extensible thanks to plugins-architecture.
- Near real-time indexing: Solr takes advantages of Lucene's Near Real-Time indexing capabilities to make sure that data is always available.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Apache	**License:** Apache 2.0

6.4.5.3 Lucene

Presentation

Apache Lucene (https://lucene.apache.org/core/) is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform.

Main concepts and features

Lucene offers powerful features through a simple API:

- Scalable and High-Performance Indexing: It offers incremental indexing as fast as batch indexing.
- Powerful, Accurate and Efficient Search Algorithms: It offers a ranked searching, many powerful query types, fielded searching. It has a multiple-index searching with merged results and allows simultaneous update and searching. The storage engine can be configured.
- Cross-Platform Solution: available as Open Source software under the Apache License which allows Lucene to be used in both commercial and Open Source programs.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Apache	**License:** Apache 2.0

6.4.5.4 Comparison of Search Engine Software

Table 4: Comparison of search Engine Software

Criteria/ software	Elasticsearch	Solr	Lucene
Scope	Elasticsearch is a distributed web server build over Lucene.	Solr is s a web application built around Lucene.	Lucene is a java library that can be included in a project and refer to its functions using function calls.
Node discovery	Internal Zen Discovery.	Based on zookeeper.	Not supported.
Shard placement	Dynamic, shards can be moved on demand depending on the cluster state.	Static in nature, requires manual work to migrate shards.	Inverted index, a keyword-centric data structure.
caches	Per segment, better for dynamically changing data.	Global, invalidated with each segment change.	Allows for some low-level caching of certain data structures.
Analytics Engine	Offers highly flexible aggregations.	Provides facets and powerful streaming aggregations.	Enables uses to define analytics functions.
Search Speed	Good for rapidly changing data, because of per-segment caches.	Best for static data, because of caches.	Provides ranked searching, best results returned first.
Performance	Exactness of the results depends on data placement.	Great for static data with exact calculations.	Appropriate choice in case of embed search functionality into an application.
Full text Search features	Single suggest API implementation, highlighting rescoring.	Multiple suggesters, spell checkers, rich highlighting support.	Memory-efficient and typo-tolerant suggesters.
Data Handling	Natural support with nested and object types.	Nested documents and parent-child support.	Document is the unit of search and index.
Machine learning	Commercial feature, focused on anomalies and outliers and time-series data.	Built-in on top of streaming aggregations and focused on logistic regression and learning to rank documents.	Not supported.
Query	JSON.	JSON, XML, or URL parameters.	Has its own mini-language for performing searches. Supports many powerful query types: phrase queries, wildcard queries, proximity queries, range queries and more.

6.4.6 Data Usage

6.4.6.1 Kibana

Presentation

Elastic Kibana (https://www.elastic.co/products/kibana) is an open source data visualization plugin for Elasticsearch. It provides visualization capabilities on top of the content indexed on an Elasticsearch cluster. Users can create bars, lines and scatter plots, or pie charts and maps on top of large volumes of data. The combination of Elasticsearch, Logstash, and Kibana is available as products or service. Logstash provides an input stream to Elastic for storage and search, and Kibana accesses the data for visualizations such as dashboards.

Main concepts and features

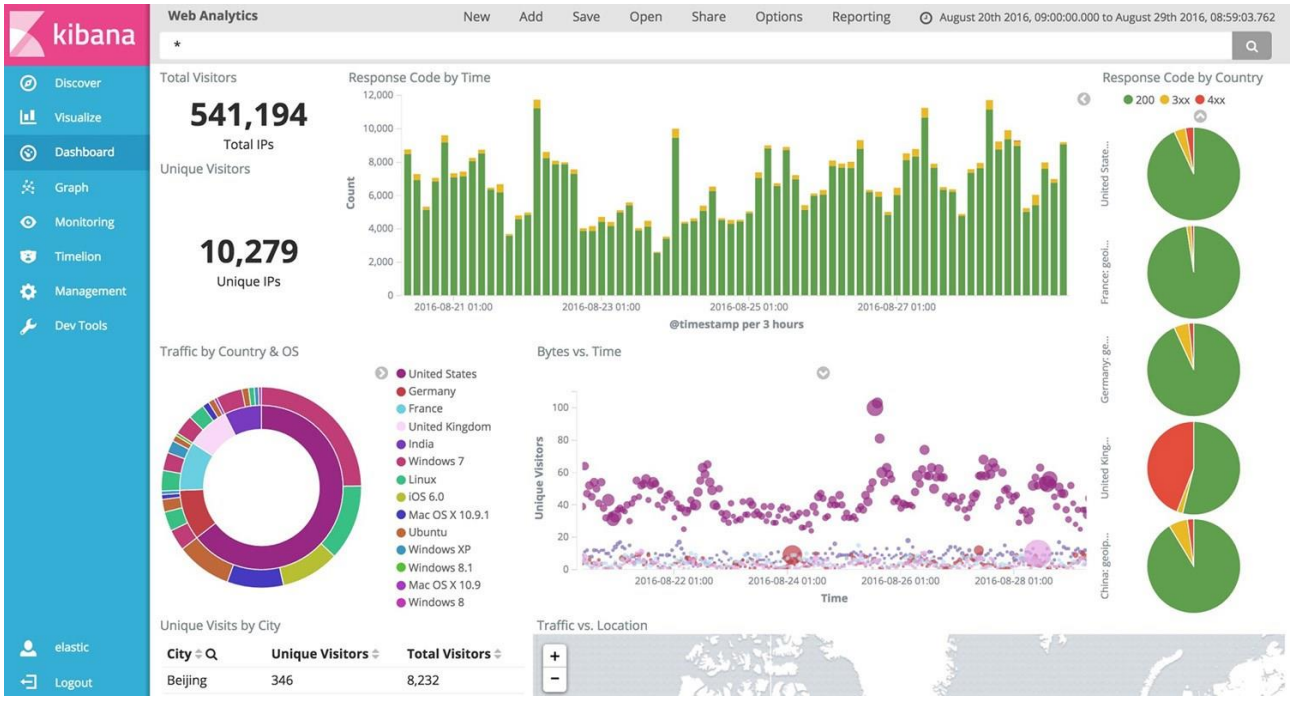

Figure 34: Kibana interface

The key features of Kibana are:

- Leverage the Elastic Maps Service to visualize geospatial data and visualize custom location data on a schematic of user' choosing.
- Perform advanced time series analysis on Elasticsearch data with curated time series UIs. Describe queries, transformations, and visualizations with powerful, easy-to-learn expressions.
- Take the relevance capabilities of a search engine, combine them with graph exploration, and uncover the uncommonly common relationships in Elasticsearch data.
- Detect the anomalies hiding in Elasticsearch data and explore the properties that significantly influence them with unsupervised machine learning features in X-Pack.
- Shareable with embed dashboards and links: dashboards can be inserted into an enterprise's internal wiki or webpage or URL is sent to co-workers to a dashboard.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Elasticsearch	**License:** Apache 2.0

6.4.6.2 Grafana

Presentation

Grafana (https://grafana.com/) is the leading open source project for visualizing metrics. Supporting rich integration for every popular database like Graphite, Prometheus, InfluxDB, and Elasticsearch. Grafana manages users, roles and organizations. through LDAP, Basic Auth and Auth Proxy. It annotates graphs with rich events from data sources including Elasticsearch, Graphite and InfluxDB.

Main concepts and features

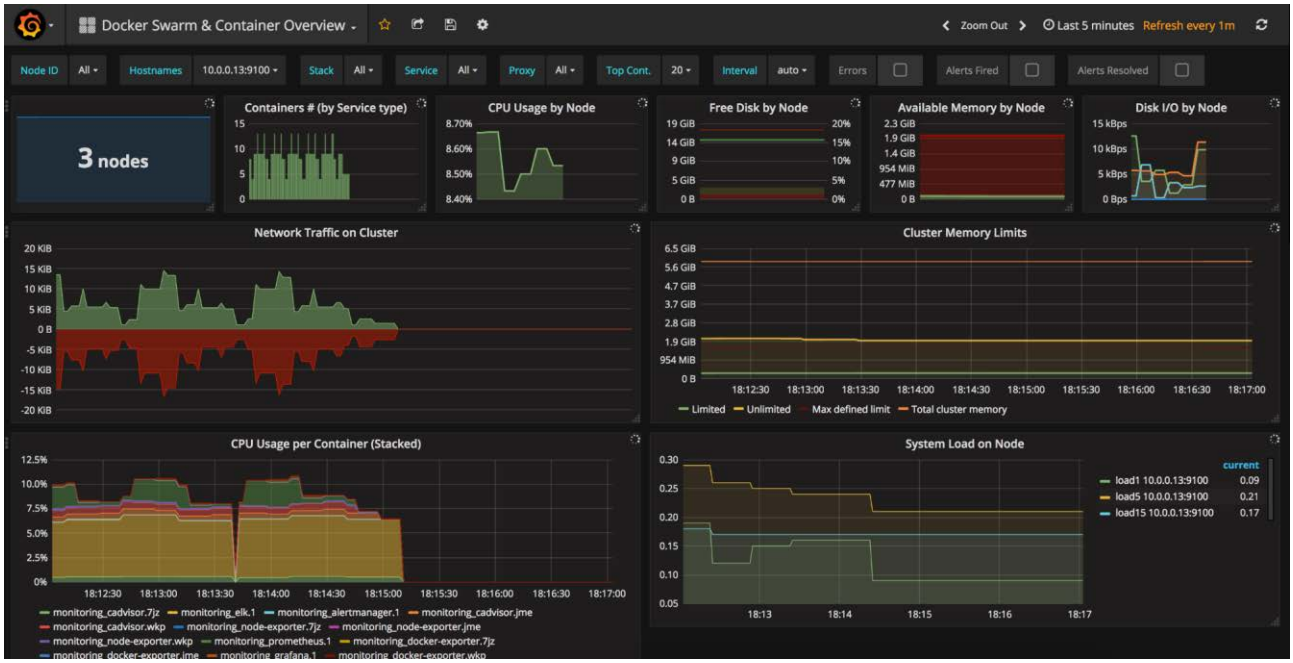

Figure 35: Grafana dashboard

Grafana allows to query, visualize, alert on and understand your metrics no matter where they are stored. Create, explore, and share dashboards with your team and foster a data driven culture:

- Visualize: Fast and flexible client-side graphs with a multitude of options. Panel plugins for many different ways to visualize metrics and logs.
- Alerting: Visually define alert rules for your most important metrics. Grafana will continuously evaluate them and can send notifications.
- Notifications: When an alert changes state it sends out notifications. Receive email notifications or get them from Slack, PagerDuty, VictorOps, OpsGenie, or via webhook.
- Dynamic Dashboards: Create dynamic and reusable dashboards with template variables that appear as dropdowns at the top of the dashboard.
- Mixed Data Sources: Mix different data sources in the same graph. A data source can be specified on a per-query basis. This works for even custom data sources.
- Annotations: Annotate graphs with rich events from different data sources.
- Ad-hoc Filters: Ad-hoc filters allow to create on the fly new key/value filters, which are automatically applied to all queries that use that data source.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Grafana	**License:** Apache 2.0

6.4.6.3 Comparison of Visualization Software

Table 5: Comparison of Data Usage Software

Criteria/ Tool	Kibana	Grafana
Data sources	It presents data from constant streaming sources, such as sensors and metric reporting.	It focuses on visualizing time-series charts based on metrics such as CPU and I/O utilization.
Data visualization	It offers large number of built-in types of charts. Users can share dashboards.	It shows data mainly though graphs but supports other ways to visualize data through a pluggable panel architecture.
Integration of data sources	Kiabana supports a native integration within the elastic stack (elasticsearch, logstash, beats).	Grafana supports many different storage backends such as influxDB, Graphite, openTSDB, and elastic stack.
Access control	Dashboards are public.	Grafana ships with role-based access.
Community	Active community.	Active community.

6.5 Monitoring

6.5.1 Prometheus

Presentation

Prometheus (https://prometheus.io/) is an open-source monitoring solution. It implements a highly dimensional data model. Prometheus has multiple modes for visualizing data: a built-in expression browser, Grafana integration, and a console template language. Existing exporters allow bridging of third-party data into Prometheus. As examples: system statistics, as well as Docker, HAProxy, StatsD, and JMX metrics.

Main concepts and features

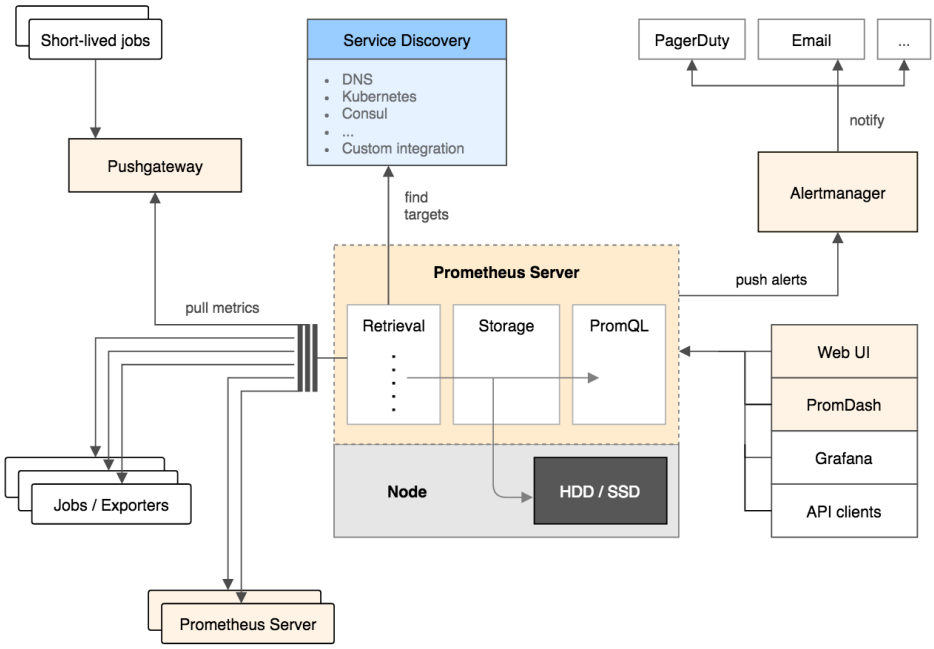

Figure 36: Prometheus Architecture

Figure 36 describes the architecture of Prometheus and some of its ecosystem components.

- The main Prometheus server which scrapes and stores data.
- A push gateway responsible of supporting short-lived jobs.

- client libraries for instrumenting application code. They let users define and expose internal metrics via an HTTP endpoint on their application's instance.
- An alert manager to handle alerts.

Prometheus fundamentally stores all data as time series: streams of timestamped values belonging to the same metric and the same set of labelled dimensions. Besides stored time series, Prometheus may generate temporary derived time series as the result of queries.

The Prometheus client libraries offer four metrics types:

- Counter: a counter is a cumulative metric that represents a single numerical value that only ever goes up. A counter is typically used to count requests served, tasks completed, errors occurred, etc.
- Gauge: a gauge is a metric that represents a single numerical value that can arbitrarily go up and down. Gauges are typically used for measured values like temperatures or current memory usage, but also "counts" that can go up and down, like the number of running goroutines.
- Histogram: a histogram samples some observations (usually things like request durations or response sizes) and counts them in configurable buckets. It also provides a sum of all observed values.
- Summary: similar to a histogram, a summary samples some observations. In addition, it provides a total count of observations and a sum of all observed values, it calculates configurable quantiles over a sliding time window.

Prometheus scrapes metrics from instrumented jobs, either directly or via an intermediary push gateway for short-lived jobs. It stores all scraped samples locally and runs rules over this data to either aggregate and record new time series from existing data or generate alerts. Grafana or other API consumers can be used to visualize the collected data.

Prometheus' main features are as:

- a multi-dimensional data model (timeseries defined by metric name and set of key/value dimensions);
- a flexible query language to leverage this dimensionality;
- no dependency on distributed storage; single server nodes are autonomous;
- timeseries collection happens via a pull model over HTTP;
- pushing timeseries is supported via an intermediary gateway;
- targets are discovered via service discovery or static configuration;
- multiple modes of graphing and dashboarding support;
- support for hierarchical and horizontal federation.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Cloud Native Computing Foundation	**License:** Apache 2.0

6.5.2 Netdata

Presentation

Netdata (https://my-netdata.io/) is a system for distributed real-time performance and health monitoring. It provides unparalleled insights, in real-time, of what is happening on the system (including applications such as web and database servers), using modern interactive web dashboards. Netdata is fast and efficient, designed to permanently run on all systems (physical and virtual servers, containers, IoT devices), without disrupting their core functions.

Main concepts and features

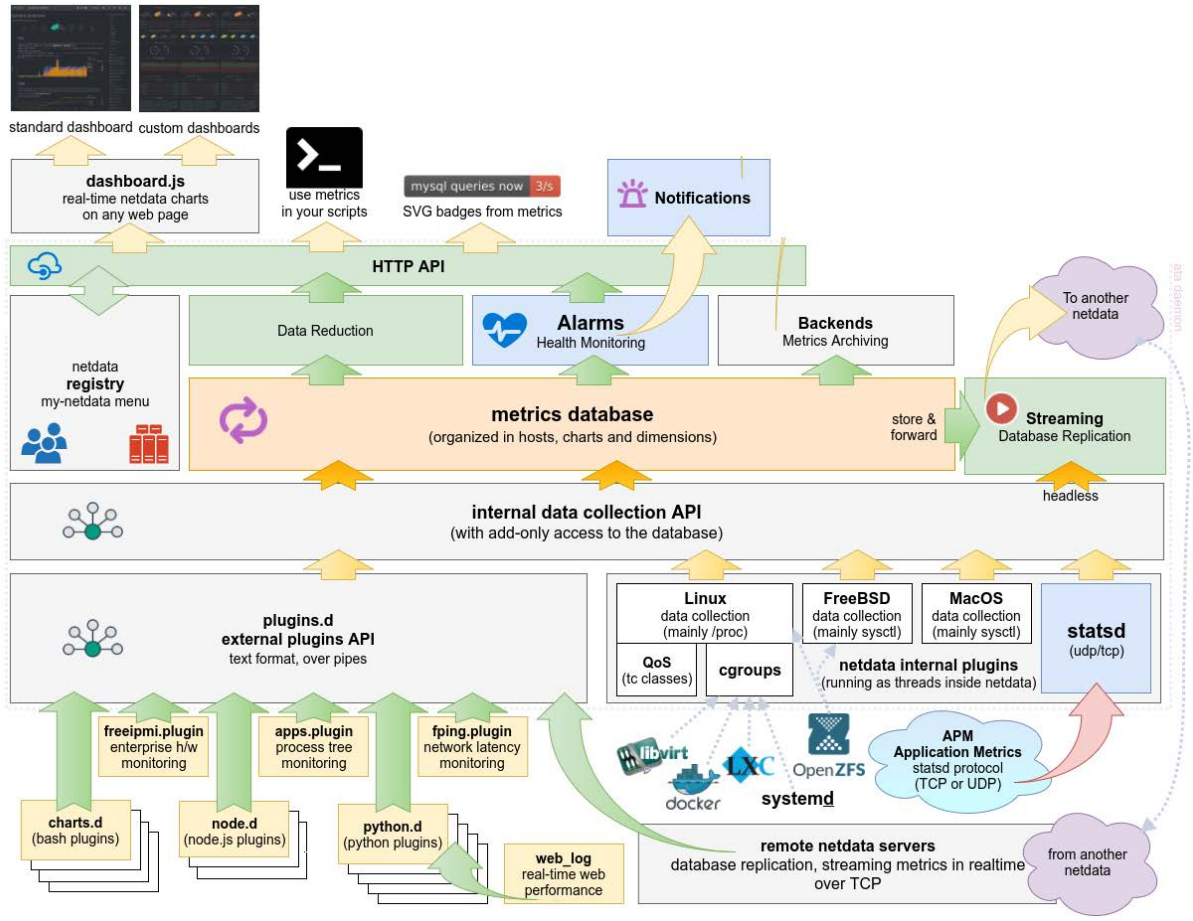

Figure 37: Netdata High-level features and Architecture

The Netdata main features are as follows:

- Interactive bootstrap dashboards: mouse and touch friendly, in 2 themes: dark, light.
- Fast response to all queries in less than 0,5 ms per metric, even on low-end hardware.
- Highly efficient: collects thousands of metrics per server per second, with just 1 % CPU utilization of a single core, a few MB of RAM and no disk I/O at all.
- Hundreds of alarms, supporting dynamic thresholds, hysteresis, alarm templates, multiple role-based notification methods (such as email, slack.com, flock.com, pushover.net, pushbullet.com, telegram.org, twilio.com, messagebird.com, kavenegar.com).
- Extensibility: users can monitor anything that they can get a metric for, using its Plugin API (anything can be a netdata plugin, BASH, python, perl, node.js, java, Go, ruby, etc.).
- Embeddability: run anywhere a Linux kernel runs (even IoT) and its charts can be embedded in web pages.
- Customizable: custom dashboards can be built using simple HTML (no javascript necessary).
- Zero configuration: auto-detects everything, it can collect up to 5 000 metrics per server out of the box.
- Zero dependencies: it is even its own web server, for its static web files and its web.
- Zero maintenance: users just run it, it does the rest.
- Scalability to infinity: requiring minimal central resources.

- Several operating modes: autonomous host monitoring, headless data collector, forwarding proxy, store and forward proxy, central multi-host monitoring, in all possible configurations. Each node may have different metrics retention policy and run with or without health monitoring.

Status in the Open Source Community

Readiness: TRL-9	**Ecosystem:** Netdata	**License:** GPL v3+

6.5.3 Comparison of Monitoring Software

Table 6: Comparison of Monitoring software

Criteria/ Software	Prometheus	Netdata
Scope	Prometheus is a full monitoring and trending system that includes built-in and active scraping, storing, querying, graphing, and alerting based on time series data. Prometheus can scrape multiple servers with Netdata running and store the data on a Central server, thus user can see metrics from multiple servers.	Netdata is an application which does realtime performance and health monitoring of servers and applications. It provides an HTTP api which can be used by Prometheus to collect the metrics which Netdata has collected from machines.
Data Model	It offers multi-dimensional data model and encodes dimensions explicitly as key-value pairs, called labels, attached to a metric name.	It submits metrics in time series data model.
Storage	Prometheus stores time series data on local disk.	Netdata supports backends for archiving the metrics or providing long term dashboards such as Graphite, openTSDB, Prometheus.

7 Standards support to IoT Virtualization

7.1 Introduction

The IoT Standards are constantly evolving in number, in scope, in application domains, etc. It is important to know which Standards are applicable to the IoT community. There are two kinds of such standards:

- Standards that are IoT-specific and have been or are being developed within SDOs or SSOs that deal specifically with IoT issues. The SDOs and SSOs involved can be general purpose and have IoT-specific work groups.
- General purpose standards than can apply also to the IoT domain. A good example of such standards are the security standards: most of these standards are developed for a large range of systems, possibly without any IoT part.

From this standpoint, IoT Virtualization will benefit from the advances in Cloud Computing standardization.

A key question, taking into account the purpose and content of the present document, is how far the work done in the Open Source communities and the Standards communities are connected (and, even better, coordinated).

7.2 Standards Landscapes for IoT Virtualization

7.2.1 An initial list of IoT Standards from AIOTI

The work done

The AIOTI Work Group 3 on Standardization has developed an IoT landscape (see ETSI TR 103 375 [i.5]) using the distinction between the horizontal and vertical domains for the classification of the organizations that are active in IoT standardization. The classification of IoT standardization organizations has been done along two dimensions:

- Vertical domains representing 8 sectors where IoT systems are developed and deployed.
- An "horizontal" layer that groups standards that span across vertical domains, in particular regarding telecommunications.

In order to give an indication of the relative importance of "horizontal" versus "vertical" standards, the ETSI report on the IoT Landscape (see ETSI TR 103 375 [i.5]) has identified 329 standards that apply to IoT systems. Those standards have been further classified in:

- 150 "Horizontal" standards, mostly addressing communication and connectivity, integration/interoperability and IoT architecture.
- 179 "Vertical" standards, mostly identified in the Smart Mobility, Smart Living and Smart Manufacturing domains.

In addition to the standards identified in ETSI TR 103 375 [i.5], a set of gaps have also been identified (see ETSI TR 103 376 [i.6]). They are, in some cases, a source of new work to be done in the IoT Standardization.

Impact on/from IoT Virtualization

The standardization in the IoT community has little impact on Cloud Computing standardization. The basis standards that are common to both IoT and Cloud Computing are generic ones, that are not specific to IoT. An example of such standards is those related to security: all that applies in IoT is also applying in Cloud Computing.

7.2.2 A landscape of Cloud Computing Standards

The work done

The European Commission, together with ETSI have been launching two steps of the "Cloud Standards Coordination" (CSC) with a Phase 1 in 2013 and a Phase 2 that has concluded in February 2016. Both in Phase 1 and in Phase 2, a standards landscaping has been conducted.

The Landscaping in CSC Phase 2 has been an opportunity to see the progress in Cloud Computing standardization. In the final report of CSC Phase 2 landscaping (see ETSI SR 003 392 [i.7]):

- There are 114 documents from 16 organizations, 94 with the status "Published", 14 with the status "Draft" and 6 with the status "In progress". This is to be compared with the list of CSC phase 1 that included 65 documents from 17 organizations, 50 with the status "Published" and 15 with the status "Draft".
- In the feedback gathered from the Cloud Computing Users, they indicate Service Level Agreements (SLA), Interoperability and Security as their Top 3 concerns. The expectation is that standardization should continue its efforts in support pf the resolution of those concerns (i.e. in the closing of the corresponding standards "gaps").

Impact on/from IoT Virtualization

The most recent evaluation of the progress in Cloud Computing standardization tends to show a shift within the Cloud Computing community towards the resolution of "gaps" via the development of specific Open Source Software components available for the developers of Cloud Computing applications (and in particular Cloud-Native applications). Consequently, the number of available standards remains stable and a large part of the efforts towards interoperability are dedicated to interoperability between OSS components.

7.3 Recent advances in IoT Standardization

7.3.1 Introduction

The development of IoT Virtualization introduces a new approach into dealing with some technical questions whose origin may be from the IoT domain as well as the Cloud Computing domain. Amongst the technical issues that are handled both by the OSS communities and the standardization community in both domains, two seems of significant importance: Big Data and the handling of massive amounts of data in a Cloud-Native manner; and Semantic Interoperability with the expectation that interoperability can be handled at higher-levels of the "interoperability stack".

7.3.2 Big Data

The work done

Big Data is a key component to IoT Virtualization solutions that address the need to handle massive amounts of data in a Cloud-Native manner. The Big Data solutions have originated in (Massively) Distributed Computing and flourished in the Cloud Computing space. The overall approach has been in the development of technical implementations (backed by a set of very strong conceptual and architectural considerations) rather than by the definition of standardized approaches. The usual pattern was the development of an initial solution (e.g. MapReduce) and its further enlargement by making it available to the Open Source Community.

Only relatively recently, some attempts have been made by the Standards community in trying to rationalize and, whenever possible, to standardize some elements (e.g. vocabularies or terminologies). Some of the most relevant and recent attempts from the global Standards community regarding Big Data are analysed below:

- NIST Big Data Framework.
  The NIST has worked on the definition of a Big Data Interoperability Framework (NBDIF) that has produced 7 final documents developed within five subgroups:
  - NIST Big Data Definitions & Taxonomies (NIST SP 1500-1 [i.10] and NIST SP 1500-2 [i.11])
  - NIST Big Data Use Case & Requirements (NIST SP 1500-3 [i.12])
  - NIST Big Data Security & Privacy (NIST SP 1500-4 [i.13])
  - NIST Big Data Reference Architecture (NIST SP 1500-5 [i.14] and NIST SP 1500-6 [i.15])
  - NIST Big Data Technology Roadmap (NIST SP 1500-7 [i.16])
- Recommendation ITU-T Y.4114 [i.19].
  ITU-T has worked on a framework specifically addressing the IoT with Big Data. This has been translating in the publication in July 2017 of the Y.4114 Recommendation on "Specific requirements and capabilities of the Internet of things for big data". This Recommendation complements the developments on common requirements of the IoT described in Recommendation ITU-T Y.4100/Y.2066 [i.17] and the functional framework and capabilities of the IoT described in Recommendation ITU-T Y.2068 [i.20] in terms of the specific requirements and capabilities that the IoT is expected to support in order to address the challenges related to big data.
- ISO/IEC JTC 1 Work Group 9.
  The Work Group 9 on Big Data serves as the focus of the Big Data standardization program of JTC 1. WG9 will identify standardization gaps and develop foundational standards - including reference architecture and vocabulary - that will guide the way for other big data efforts both within WG 9 and throughout JTC 1.
  ISO/IEC JTC1 WG9 is working on two documents:
  - ISO/IEC DIS 20546 [i.18]: "Information technology -- Big data -- Overview and vocabulary".
  - ISO/IEC 20547 [i.21]: "Information technology -- Big data reference architecture".

Impact on/from IoT Virtualization

The works described above are having a global approach of defining a reference framework that the practitioners in the IoT community (e.g. industry) may use in the specification and design of IoT systems dealing with massive handling of data.

7.3.3 Semantic Interoperability

The work done

Ensuring interoperability in IoT systems requires a that the interactions between their different key components be handled by looking at both the *syntax* and the *semantics* of the interfaces of these components. Syntactic interoperability can be obtained through clearly defined and agreed upon data and interface formats. Semantic interoperability can be achieved through commonly agreed information models defined with ontologies of the terms used as part of the interfaces and exchanged data.

Some of the most relevant and recent attempts from the global Standards community regarding Big Data are analysed below:

- SAREF
  The Smart Appliances REFerence ontology (SAREF) is the result of an EU initiative launched in 2013 with the support of ETSI. SAREF is an addition to existing communication protocols to enable the translation of information coming from existing (and future) protocols to and from all other protocols that are referenced to SAREF. The initial focus was on the optimization of energy management in smart buildings

  The first resulting semantic model - SAREF - was standardized by ETSI in November 2015 (see [i.8] for the most recent version). SAREF is a first ontology standard in the IoT ecosystem, it offers an approach and a basis for the development of similar standards for potentially all other verticals. Since its first release, SAREF continues to evolve systematically into a modular network of standardized semantic models, with additional extensions: SAREF for Energy, SAREF for Environment and SAREF for Buildings; as well as on-going work in a number of other domains such as Smart Cities, Smart AgriFood, Smart Industry and Manufacturing, Automotive, eHealth and Wearables. SAREF could become a "Smart Anything REFerence ontology", enabling a better integration of semantic data from various vertical domains.

- Web of Things
  The Web of Things (WoT by W3C) focuses on the role of Web technologies for a platform of platforms as a basis for services spanning IoT platforms from microcontrollers to cloud-based server farms. Shared semantics are essential for discovery, interoperability, scaling and layering on top of existing protocols and platforms.

  For this purpose, metadata can be mainly classified as things considered as virtual representations (objects) for physical or abstract entities. Things are defined as having events, properties and actions, as a basis for easy application scripting, assuming a clean separation between the application and transport layers, which simplifies scripting. In addition, communications metadata allows servers to identify how to communicate with other servers.

  Thing descriptions are expressed in terms of W3C's resource description framework (RDF). This includes the semantics for what kind of thing it is, and the data models for its events, properties and actions. The underlying protocols are free to use whatever communication patterns are appropriate to the context according to the constraints set by the given metadata.

Impact on/from IoT Virtualization

The approaches mentioned above are two ways to approach the handling of metadata. From the point of view of the developers of Virtualized IoT applications, the underlying approaches and models may have different levels of perceived complexity. A challenge for the SAREF is to ensure that a global approach that captures many different ontologies in a single model can be fully grasped and endorsed within the supporting OSS communities.

7.4 Advances from IoT Research

Amongst the efforts developed in pre-standardization, only an unknown (but possibly small) number may become relevant references to the IoT community: only time will tell. Two of them are presented below that could have an impact on the future of IoT systems and, in particular, Virtualized IoT systems. Both are related, more or less directly, to the question of how interoperability can be supported by higher-level constructs, than offer in particular a more dynamic support to Application Programming Interfaces. The OSS communities tend to develop APIs that are very seldom standardized and may evolve in more or less controlled manner: a more dynamic support for APIs (in particular by platforms that support these dynamicity constructs) may help developers in selecting appropriate components.

Patterns of Interoperability

Syntactic and semantic interoperability have already been discussed above. Effective – and in particular more dynamic - support mechanisms may help the IoT application developers and ensure that they can effectively use supporting standards proposed by the industry. In Figure 38, five generic interoperability patterns, numbered from I) to V) have been identified that apply to systems in general, and to IoT systems in particular (see [i.9]). These patterns may be seen from two angles: one related to standards and which kind of support they provide; and one related to platforms and the definition of efficient IoT platforms that can support one or more of these patterns.

The five patterns (described in below) are:

I. Cross-Platform Access. The basic pattern where an application can interoperate with several platforms.

II. Cross-Application Domain Access. This pattern expands the previous with the ability to interoperate with platforms in different domains.

III. Platform Independence. The same application or service can be used on top of two different IoT platforms (e.g. in different regions) without changes.

IV. Platform-Scale. With this pattern, the focus is on integrating platforms of different scale.

V. High-Level Service Facades. This pattern extends the interoperability requirements from platforms to higher-level services where not only platforms but also services offer information and functions via the common API.

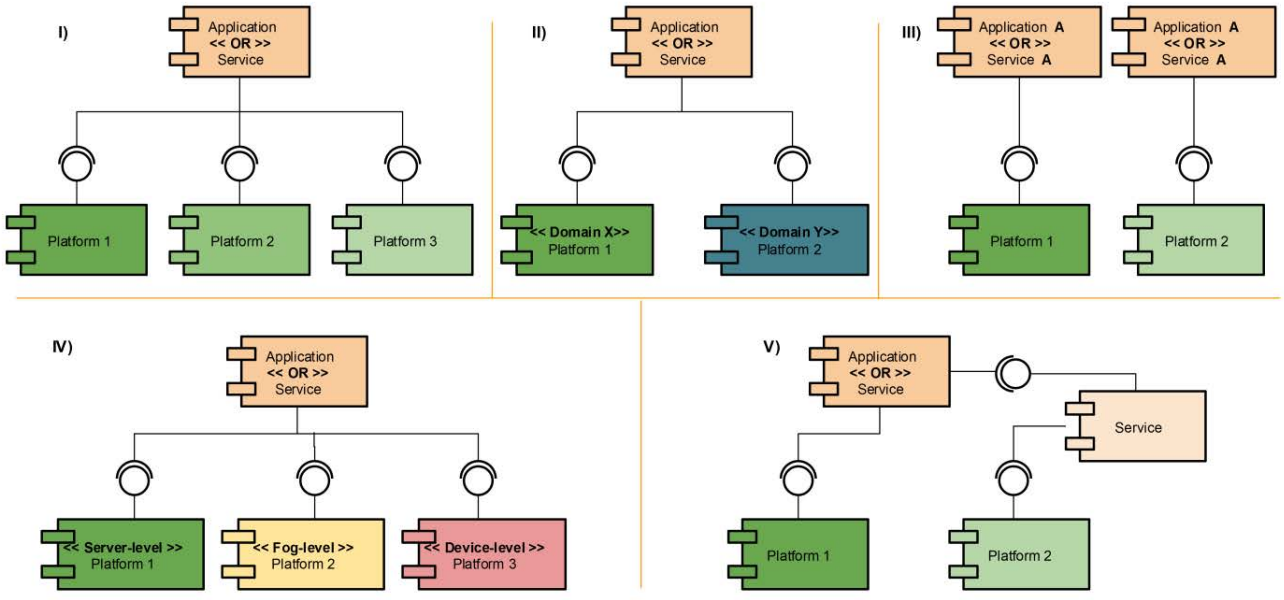

Figure 38: Five patterns of interoperability

Market Places and APIs

A new breed of IoT systems based on layered, potentially cloud-based or edge-enabled architectures is emerging with strong requirements on the connectivity between actors (e.g. sensors, gateways, platforms, data processing and analytics functions, etc.) that require complex interoperability schemes.

In this new world, IoT systems and application developers would expect that the myriad of devices that are deployed and connected to the network can seamlessly interoperate with a large range of platform services (e.g. data analytics, monitoring, visualization, etc.) and end-user/end-customers applications. With the use of the proper services, useful data and information can be exchanged between all actors across the system.

In this model, the actors can be seen as consumers and providers within an emerging application market. An IoT marketplace is a new platform to extends the "traditional" IoT platforms with brokerage concepts supporting automated discovery, trading and even pricing. Within an IoT marketplace platform, the IoT device owners will have the possibility to selectively grant access and trade their data with many potential vendors thus creating an environment where innovative solutions can be monetized (and efficiently developed) in support of a multi-vendor and multi-owner environment.

The marketplace architectures are in general supported by:

- The publication of a number of Application Programming Interfaces (APIs) that hide the actual underlying provision of the service from the consumer of the service. The implementation of the service can change without impacting the rest of the system and the evolution of the APIs can be mastered via the publication mechanism.
- An approach based on Microservices where any service (whichever its size and scope) can be published and consumed. This approach provides system flexibility; supports lean software principles; and allows fast adaptation to support emerging standards without impacting the whole system architecture.

8 Conclusions

8.1 Assessment and Lessons Learned

The first lesson learned from the analysis of the Open Source Software (OSS) landscape is that OSS is playing a crucial role of OSS in the progress of IoT Virtualization. To those who want to go this route, there is no alternative strategy with similar benefits.

The main reasons for this are addressed below:

- There is a vast amount of Cloud-native friendly OSS components available for the developers of virtualized IoT systems. Most of these components have been and are developed in the context of Cloud Computing and are able to support a very large set of features that may boost the effectiveness and rapidity of the deployment of IoT systems;
- The available OSS components have been developed, tested and validated through the work of open source communities that use the most state-of-the-art technologies for software engineering, and have set-up processes to ensure that the best skilled engineers contribute to the technical progress.
- All the OSS components identified in the present document (that include those used for the Proof-of-Concept described in ETSI TR 103 529 [i.2]) have been evaluated with the highest level of Technology Readiness (TRL-9). All these components have been validated and constantly improved by a massive usage in a very large set of top-notch systems.
- Moreover, the selected OSS components are developed by an open community with several ecosystems that work with approaches on openness, transparency, evolution, etc. that are very similar to those used in the production of standards.

It is important, though, not to downplay the role of standards in IoT virtualization: there is no contradiction between standards - that are important in the definition of IoT system architecture - and the OSS implementation. This will be reflected in the guidelines below.

8.2 Guidelines and Recommendations

8.2.1 Guidelines to designers and developers

One of the risks that virtualized IoT systems developers may face would be coming from a massive usage of OSS components without a clear adherence to specific standards. There is a well-known approach to the design a good (IoT) system: well defined architecture, adherence to standards, and an effective implementation support. A Microservice based architecture approach is perfectly suited to follow this approach and provide best-in-class systems.

Virtualized IoT systems designers and developers should take into consideration the following aspects:

- Define standards-based architectures (e.g. oneM2M) that will be more easily implemented by the OSS tool box that will enable the non-functional properties of the system (support of scalability, reliability, etc.).
- Focus on the application logic and leave the rest to the underlying common services that will support all the non-functional needs (via the OSS components).
- Avoid eco-system lock-in. Start the definition of system with the use of OSS components coming from "open ecosystems". A top-down approach and not a bottom-up from VM up to monitoring. Docker allows portability by allowing transparent use of the CSP but also by being supportive of different OSS eco-system.

8.2.2 Recommendation to oneM2M

The following recommendation is made to oneM2M in order to help the adoption of oneM2M in the virtualized IoT architectures:

> Extend oneM2M with an additional communication binding for "Apache Kafka" in order to better deal with use cases demanding high-throughput.

8.2.3 Recommendation to AIOTI and the IoT community

The IoT community - and in particular the AIOTI - could find a way to adopt a list of "IoT-friendly" OSS components (possibly based on the list provided in the present document) and to maintain for some time in order to assist the IoT systems designers that consider a transition to IoT virtualization.

The inclusion in such a list could be based on the adherence on basic principles such as a maximum Technology Readiness Level (TRL-9), transparent processes regarding the selection of features for development, etc.

Annex A:
Change History

Date	Version	Information about changes
Dec. 2017	0.1.0	Early version for discussion at January 9th, 2018 SmartM2M TC Meeting
Mar. 2018	0.2.0	First early stable version uploaded on the SmartM2M TC portal
Mar. 2018	0.2.1	Cleaned-up version for the OSS Components part
Mar. 2018	0.3.0	First stable version uploaded on the SmartM2M TC portal for March 27th ad-hoc meeting
May 2018	0.9.0	Final draft for internal review by STF 535 before submission to SmartM2M participants
July 2018	0.9.1	ETSI Secretariat Check and EditHelp clean-up

History

Document history		
V1.1.1	August 2018	Publication